Spring 1992

Monitoring Computer Systems: An Intelligent Approach

Myron Zhihong Xu
Old Dominion University

MONITORING COMPUTER SYSTEMS:

AN INTELLIGENT APPROACH

by

Myron Zhihong Xu
B.S. December 1982, Shanghai Industry University
M.S. August 1989, Brigham Young University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May, 1992

Approved by:

Dr. Stewart N. Shen (Director)

Dr. Ravi Mukkamala

Dr. Larry Wilson

Dr. Jen Kuang Huang

ACKNOWLEDGEMENTS

I am deeply grateful to my dissertation director and academic advisor, Dr. Stewart N. Shen, for his patient advice and generous support during my PhD program. Without his guidance to my research, the completion of this dissertation would not be possible. A special thanks goes to my dissertation committee members, Drs. Ravi Mukkamala, Larry Wilson, and Jen Kuang Huang. Their expertise, suggestions, and assistance during the various stages of this research were invaluable.

I greatly appreciate the help from my fellow students in the AI research laboratory at the Computer Science Department of Old Dominion University.

Finally, I wish to dedicate this dissertation to my family. Their values and dedication throughout my education have instilled in me an unquenchable thirst for knowledge. Above all, my heartfelt gratitude is reserved for my parents.

Contents

List of Figures

ABSTRACT

MONITORING COMPUTER SYSTEMS: AN INTELLIGENT APPROACH

Myron Zhihong Xu
Old Dominion University, 1992
Director: Dr. Stewart N. Shen

Monitoring modern computer systems is increasingly difficult due to their peculiar characteristics. To cope with this situation, the dissertation develops an approach to intelligent monitoring. The resulting model consists of three major designs: representing targets, controlling data collection, and autonomously refining monitoring performance. The model explores a more declarative object-oriented model by introducing virtual objects to dynamically compose abstract representations, while it treats conventional hard-wired hierarchies and predefined object classes as primitive structures. Taking the representational framework as a reasoning bed, the design for controlling mechanisms adopts default reasoning backed up with ordered constraints, so that the amount of data collected, levels of details, semantics, and resolution of observation can be appropriately controlled. The refining mechanisms classify invoked knowledge and update the classified knowledge in terms of the feedback from monitoring. The approach is designed first and then formally specified. Applications of the resulting model are examined and an operational prototype is implemented. Thus the dissertation establishes a basis for an approach to intelligent monitoring, one which would be equipped to deal effectively with the difficulties that arise in monitoring modern computer systems.

Chapter 1

Introduction

Motivated by challenges resulting from certain characteristics of modern computer systems, this thesis explores a new approach to monitoring computer systems. The introductory chapter states the intention of the thesis by clarifying several aspects of monitoring computer systems: its importance, fundamental approaches, recently confronted challenges, and consequent issues that ought to be addressed in this area. The chapter then outlines the organization of the thesis.

1.1 Importance of Monitoring

There are essentially two ways to facilitate performance analysis: modeling and monitoring [59]. While modeling has important applications, it can become intractable for complex systems unless overly simplistic assumptions are made. In many cases, modeling may not be sufficiently accurate to exhibit system performance. As an alternative, analysts often rely on monitoring tools for performance analysis [55, 4, 58, 40].

Over the years, monitoring has become an essential function integrated into computer systems. In a study of program developmental tools [57], a quarter of those tools were highly dependent on monitoring information, including those under the cat-

egories of tracing, tuning, timing, and resource allocation. As summarized in [34, 57], monitoring is a fundamental component of the following computing activities:

- One use of monitoring is to facilitate the debugging of complex programs. People often wish to use a monitoring system partially as a diagnostic system [38].
- Another use is to ensure the efficient use of limited computing resources. In particular, software resources have become a main concern in analyzing the utilization of a computer system.
- Monitoring is also used to query a computer system—not for performance measures but merely for status information.
- Monitored information may additionally be used by application programs for load balancing and graceful degradation in the presence of hardware and software failures.

The facilities comprising the above activities are more frequently utilized by computer users—the majority of whom are now application-oriented users—as more computers feature parallel and distributed computations.

1.2 Types of Monitoring

Monitoring is the process by which the data that characterizes the workload and performance indices of a system is provided. Collected data can be classified into two categories: the data at the assembly language level and the data at the high language level. Not long ago, most of the features found in monitors were meant to serve for observing computing activities at the hardware level. As a result, hardware monitoring facilities are widely used as an attachment to computer systems. Recently, due to the rapid development of programming languages and the swelling population of computer users—mainly application-oriented users—demands in the field of

analyzing program execution have forced monitoring to address the level of source programming languages [59]. Gradually, monitoring activities are developing in three major ways [74]:

- The first monitors the performance of hardware resources, for example, how physical resources are utilized and how the hardware architecture of a computer system functions in a particular working environment, such as a batch processing or an interactive computation.

- The second observes activities at the system software level for the purpose of system analysis. At that level, the working load on each processor, input and output delay, memory swapping and other typical characteristics are often of concern.

- The third involves observing software performance at the application level. The execution of statements, functions, and algorithms in a program is often monitored. This also includes the observation of the performance of software facilities, such as algorithms used in a compiler for distributing local and global variables, and accessing processes employed inside a data base management system.

1.3 Challenges in Monitoring

New technology and increasing demands for improving communication and productivity spawn new architectures and protocols in computer systems designed for diverse applications. Consequently, monitoring computing activities becomes more difficult because of some distinctive qualities inherent in those systems. Specifically, three characteristics are emphasized as having a strong impact on monitoring systems:

- invoked knowledge evolves over time;

- computing activities are diverse;
- the complexity of a computer system often makes comprehensive monitoring prohibitive.

These complications suggest many needed improvements to monitoring technology so that monitoring may effectively assist in debugging and measuring computer systems [20, 22]. Desirable properties of a monitoring system include the following:

- widely applicable across types of machines,
- easily customized with respect to applications,
- little interference while capturing critical data,
- supplying effective assistance in fault-detecting,
- providing users with deliberate data,
- acting in accordance with he necessary concerns of system performance.

Among many desired properties that a monitoring tool is required to possess, four should be considered as overwhelming issues and necessitate noticeable interest in this research. The section that follows identifies these important issues and outlines their significance.

1.4 Overwhelming Issues

Four issues, considered to have the key strength in alleviating the difficulties of monitoring modern computer systems, are listed in the following:

- provide abstract representations of a monitored system with flexibility,

- integrate the controlling knowledge into abstract representations of a monitored system in such a way that both the cost and accuracy of monitoring could be effectively controlled,
- facilitate performance analysis by supplying meaningful information,
- regulate the monitoring to behave in an adaptable way.

The significance of the above issues is elaborated hereafter.

The First Issue

An abstract representation of a monitored system may help analysts understand the system. This is because the various details at each abstract layer can be isolated and decomposed into simpler cases by allowing the monitored information to be presented at an appropriate level of abstraction. High-level patterns correspond to highly aggregated, abstracted, or condensed descriptions of what are actually detailed phenomena. The idea is that the higher the level of structure to be matched or recognized, the more conceptual ground would be covered in one inferential leap [3, 14].

It has long been advocated that systems should be constructed as a hierarchy of levels in which each level is represented by a specific abstraction; then one can understand the system without having to know details of the lower levels [78]. Nevertheless, because of the difficulty in achieving acceptable consistency and descriptive capabilities for such abstraction, the complexity of computer systems makes it difficult to represent a computer system through an abstract model.

Consistency means that the abstract description can interface properly with related layers so that no contradiction exists. Descriptive capability means that necessary mechanisms are made available for effectively specifying abstract representations of computing activities at any user-preferred levels. These factors apparently make accurate abstractions for diverse computing activities more demanding and more

challenging.

The Second Issue

Because comprehensive monitoring is prohibitively difficult, the data collection ought to be selective. Consequently this requires the controlling knowledge to be attached to the part responsible for on-line data collection. One way of making this occur is to take advantage of the use of abstraction by integrating knowledge into an abstract representation of a monitored system [37, 77, 15]. The benefits may be twofold:

First, it may speed up the processing of observable data. Attached functions can be invoked immediately without searching for a function. The complexity of computer systems involves large amounts of knowledge, so it is desirable to get a suitable set of knowledge applied in time. The inherent knowledge could assist monitoring in achieving optimal performance by possibly making the filtering process occur at the monitoring stage. Early-stage filtering may effectively handle the contradictory situation: inclusive monitoring is often impossible, whereas the demands for comprehensively demonstrating a monitored system become even stronger.

Second, by letting each abstract layer have built-in controlling functions, a monitor is capable of strengthening the integration of layered representations. One consequence is that the internal organization of a system can be more easily observable by a user. This is due to the fact that not only is the user being encouraged to look only at those features that are relevant to the user's task, but also the user may actually be prevented from getting into the internal workings of the system beyond what is necessary.

Hence, organizing the controlling knowledge and the representation of a monitored system into a cohesive structure is a desirable capability.

The Third Issue

As stated early on, the complexity of computer systems makes it difficult to track the behavior of those systems manually. Computing activities vary over time, and changes may be invisible. On the other hand, in order to analyze problems that could be hidden in vague requirements or in a section of a validated program, accuracy in reflecting the behavior of an entire system is still desired. Evidently, relying on human intervention to achieve optimal performance of monitoring may not be realistic.

Therefore, it is desirable that meaningful information concerning the performance of a monitored system be provided primarily by a monitoring tool. This implies that a monitoring tool should not only be able to selectively collect monitored data, but also provide capabilities in locating a proper abstract level to be monitored, interpreting data into a meaningful and understandable manner, and providing causal data in the event of anomalies, as well as acting accordingly during observation.

The Fourth Issue

The static feature existing to monitoring systems is a barrier in monitoring a variety of computing activities, especially in a rapidly changing system [39]. There are many facts that may be convincing in this issue; the following brings to light some important reasons.

Various perspectives of activities of a computer system can usually be categorized into three types: the performance of physical resources, the performance of logical resources, and the performance of virtual resources. The utilization of the latter two types of resources ought to be measured in ways which largely depend on the concerns of eventual users; such concerns are often not predictable in the designing of monitoring tools. Thus, when an analyst looks at specific aspects of a monitored system, he may be limited by certain designs which exist only to collect predefined types of data. Such a tool will inevitably be considered inefficient [42].

The understanding of a complex system normally lacks sophistication until such a system has been in operation for an extended period of time, so that a monitor, which depends heavily on such understanding, needs to be refined. Also, one distinct characteristic of modern computer systems is their open-ended feature. This means that the behavior and even the configuration of a computer system changes with continual use. It then becomes necessary for a monitoring tool to possess adaptable behavior as well as the ability to enrich its controlling knowledge during monitoring.

Apparently, one may conclude that the more evolvable a monitoring tool is, the stronger applicability it will possess.

1.5 Organization of the Thesis

Addressing preceding issues, this thesis researches a new approach to monitoring computer systems and organizes the exploration of such an approach into the next six chapters.

Chapter 2 gives a brief survey of conventional technology and existing research projects and systems in the area of monitoring. It summarizes weaknesses in those existing monitoring systems with respect to the four important issues mentioned above. With many urgently needed improvements in mind, Chapter 2, therefore, further intends to indicate that intelligence should be greatly enhanced in a monitor system. The discussion then concludes with a proposal which states that an approach to intelligent monitoring is in demand.

Chapter 3 starts with the identification of main components, desired capabilities, and necessary distinctions in such an intelligent system. The chapter later discusses mechanisms that compose a model for intelligent monitoring. In order to sharpen the main research, the design emphasizes three aspects—knowledge representation, reasoning methodology, and knowledge acquisition—which may heavily affect properties

of an intelligent monitor.

Chapter 4 formally specifies the resulting monitoring model in terms of the previous design. It first clarifies functions embedded in each major component of the model, then concentrates on the three major parts of an intelligent monitor. The main principle guiding this specification is that the resulting model must be generic, as well as fundamental.

To assure comprehension of the developed model, Chapter 5 examines applications of the model. Computing activities involved in a file access are selected as targets for examination. Applications exhibit the capabilities of supporting intelligent monitoring with the developed model and are intended to illustrate a way of monitoring computing activities in a typical modern computer system.

Chapter 6 furnishes a prototype implementation of the presented model. This prototypical model is implemented to achieve two expectations: to make certain that mechanisms designed to work together operate as expected, and to experiment with the unavoidable technical details which call for supplementing the researched model.

Finally, Chapter 7 contains an epilogue of this dissertation, consisting of three parts: summary, contributions, and necessary future work as it relates to this research.

Chapter 2

Review of Monitoring Technology

For the purpose of clarifying what this thesis argues, the following chapter reviews monitoring technology with emphasis on existing active research projects. Given that most of these selected projects are the result of recent research, they also satisfy two prerequisites for review: one is that such projects are influential in this area, and the other is that the ideas exposed in these projects relate to this research. Intending to provide arguments for exploring new approaches to monitoring, the chapter reviews these monitoring systems in light of four issues that have been asserted as vital for a monitor to have the ability to deal with characteristics of modern computer systems. Additionally, to provide background for the survey, a few words about the conventional methodology of monitoring are given first.

2.1 Conventional Methodology

In general, there are two basic ways to monitor computer systems, namely, the hardware approach and the software/firmware approach [28, 33]. All others fall into the category that has these two as basic approaches. While Appendix A discusses in some detail the conventional methodology, a highly abstract introduction of conventional

technology is given in the following.

Conventionally, monitoring and measuring have been done with fundamental techniques in hardware engineering. One of the main advantages of hardware monitoring is that such a device can be designed to have minimal or no effect on a host system. It fits well into some aspects of run-time monitoring. However, hardware monitoring generally provides analysis with limited, low-level information and often uses sophisticated hardware features to get valuable but fixed-type information. On the other hand, software monitors can present information with regard to applications. The resulting advantages are that monitors are adaptable and portable, and that they allow users to evaluate interactively the performance history of a monitored system. The major deficiencies with the software monitoring approach are overhead, inaccuracy, and change of system behavior [8, 10]. Hence, by having alternatives in monitoring, a hybrid monitoring model may possibly take advantages from each approach as well as avoid problems involved in each.

2.2 Survey of Influential Research Projects

A brief survey of several of the most noteworthy monitoring projects is given in this section. These systems have achieved great success and have brought many good ideas to researchers in this area. However, instead of highlighting their success, this study intends to reveal some common weaknesses regarding issues proposed in the previous chapter.

2.2.1 Structural Approach to Monitoring

Svobodova at the MIT laboratory proposed that monitors should be designed as part of a system by following the principles and methods of structural system design [74]. His intention was to convey a new way of thinking about performance monitoring;

in particular, special attention was given to the relationship between performance monitoring and reliability monitoring, and to the impact of computer system design principles.

His proposal suggested that the methodology used in software engineering—such as structured programming, abstraction, and top-down hierarchical design—should be applied to the design of monitoring tools. Nevertheless, the layered abstraction in his proposal was only for hardware resources. As argued in the preceding chapter, an abstract model should not only support the monitoring of hardware and software resources, but should also consider intangible resources, such as computing algorithms and operating policies. Besides, the methodology in his proposal did not discuss the need for the integration of controlling knowledge into such abstraction. Last, his proposal did not present a concrete model that might validate his claim. Certainly, many good ideas were proposed, but most of those ideas could only be regarded as suggestions. He left readers to do some work of either theoretical design or experimental implementation of his proposal. Thus, the whole methodology remains only as a principal idea for addressing new challenges of monitoring.

2.2.2 Relational Approach to Monitoring

Snodgrass at the University of North Carolina, following his dissertation work at Carnegie-Mellon University [68], developed a methodology that applied a relational model to monitoring [69]. Snodgrass considered that historical databases were an appropriate formalization of the information processed by a monitor. Primary benefits include a consistent structure for the information and the use of powerful declarative query languages. With his approach, a user is presented with the conceptual view through which the dynamic behavior of a monitored system is seen as a collection of historical relations. By making historical queries on a conceptual database, a user is specifying, in a nonprocedural fashion, the sensors to be enabled, the analysis to be

carried out, and even the graphical presentation to be derived from observed data.

However, since controlling knowledge is not attached to the process of data collection, the monitor has no ability to collect data selectively during monitoring. The choice made on types of data to be fully monitored relies on the prediction of a user and is not changeable once the prediction is made.

Conveying meaningful information is restricted. On the one hand, sensors are enabled and the data is collected after the specification step, which allows a sensor to be activated automatically based on information from the query. On the other hand, there is no control over what information is to be collected from a target, because the monitoring has no ability to detect what portion of the information might be preferable. Hence, the collection can address a correct type of data by prediction but not the necessary part within the selected data type.

The relational model of monitoring has influenced the static features of monitoring through two means: 1) a user can use the query method to look at the performance of a certain part inside a system; 2) upon a query, some aspects of sensor installation are automated. In spite of these achievements, as Snodgrass also recognized, the specification of data to be collected is still too static. The first reason is that queries must be specified before the requested data is collected. The second is that a user must need to know *a priori* precisely what information is to be collected. The third is that a user may want to replay the display, or vary the display rate, but these functions are impossible in his approach. The fourth is that a sensor only stays for a predetermined period of time, which is defined by a user, or is forcibly terminated earlier by the controlling part of the monitor, but it can not be determined in terms of the performance of monitoring.

Furthermore, it is difficult to get a multi-dimensional view of system performance by relational queries. Although a user may relay a collection of historical relations for analysis, by means of a relational data model, the system will have problems in dealing

with nonlinear information structure. To understand a monitored system, a user is required to have sophisticated expertise in computers. For example, a user may first need to have an overall understanding of a monitored system. The user then needs to design a series of queries to form a general view. Finally, the user must have knowledge to combine collected observations from each individual monitoring. Excessive reliance on an end user may lead to poor efficiency of monitoring because it is difficult for the user to find the right place and the right type of data, as well as the time to monitor, unless the user knows the system very well. This prerequisite contradicts the purpose of monitoring, which is to help end users understand a complex system.

In Snodgrass's late work, some filtering techniques were used to select data so that the number of data packets to be collected could be greatly reduced. Nevertheless, reducing the amount of data collected occurs at the expense of cutting down monitoring dimensions by reducing its scope. This may not be desirable since sometimes one may like to reduce details or resolution in light of the significance of data but not the scope of monitoring. As Snodgrass realized, intelligence support is demanded but is left to future work. Without intelligence, efficiency depends upon human expertise or upon greater complexity introduced into a monitor. Consequently, the monitor's complexity becomes questionable. Components of the monitor include TQuel compiler, a sensor-description language translator, a sophisticated query optimizer, an incremental algebraic interpreter, and an incremental display generator. Therefore, while the relational model provided a coherent basis for all of these tools, Snodgrass acknowledged that this approach, rather than reducing complexity, generally shifted it from users to monitors.

2.2.3 Monitoring by Means of Trace Files and Interactive Analysis

Monit, developed by Kerola and Schwetman at Microeletronics and Computer Technology Corp., is for performance evaluation of parallel systems. It generates trace files while monitoring the execution of parallel programs [41]. Monit is an interactive program for a SUN-3 workstation; it processes an event file to produce statistical summaries and time-based bar graphs, and thereby gives a user insight into the performance characteristics and behavior of a high-level computing activity.

With that system, a user is able to select interactively the displayed items, resolution, and time intervals of interest. Despite these abilities, a deficiency arises along with some benefits. Since Monit itself has no ability to select or adjust the information being observed, it leaves the effectiveness of monitoring largely dependent on the off-line process in which an analyst has to pull out meaningful information from massive data collected. Two problems occur at this point. The first is that filtering is conducted at an off-line stage, and the collecting process may be unable to collect data comprehensively. As a result, the off-line process may not be able to mimic the facts because complexity of a modern system often prohibits monitoring to get all necessary data. The second problem is that without a structural representation, the combining of linear type information in order to exhibit the main features of the target could often lie beyond the capabilities of an average user. Although Monit provides more meaningful information by displaying bar charts on a Sun Window image, it is only the result of a later-stage analysis. That is, with such a monitoring tool, understanding the behavior of a system heavily depends on an analysis which is based on ill-structured and likely non-inclusive data.

Moreover, since Monit does not know how to distinguish whether some observed data may reveal more insight than others during monitoring, data is blindly collected

at the front stage. Consequently, in order to provide a sufficient amount of data for off-line analysis, either the cost of the collection is too high or the comprehensiveness of monitoring is strictly limited.

2.2.4 A Hybrid Approach to Monitoring

Haban and Wybranietz of the University of Kaiserslautern proposed a system that adopted the model combining hardware and software to monitor the performance and analysis of distributed systems [34]. Their intention was to solve two urgent problems, namely, interference from a monitoring system to a monitored system, and the presentation of meaningful data to users. They focused on several effective steps in monitoring, such as data collection, analysis, and the presentation of execution data. A special hardware support, which consisted of a test and measurement processor (TMP), was designed and implemented in the nodes of an experimental multicomputer system consisting of eleven nodes. The data collected by TMPs are passed to a central monitoring station where the easy-to-read information is generated for display. The operations of a TMP are completely transparent with a minimal (less than 0.1%) overhead of the measured system. Although the success from an attached TMP is substantial, some serious weaknesses still exist.

Since the improvement achieved in their project seems largely due to the use of the TMP components—the heavy dependence on hardware support—the result is that the portability of such a model seems questionable. Another factor is that their project overlooked the possible contribution that may come from the software part of a monitor, and it paid little attention to monitoring and debugging at high-level programs. Hence, these factors greatly reduce the applicability of effectively monitoring modern computer systems.

It is reasonable to doubt that with an additional processor, separate memory, and dedicated software, the monitor is economical to operate and easy to install and

maintain. Moreover, such a monitoring tool is hard to upgrade because it is predefined into firmware. This may be good for some types of monitoring, but in general it is not suitable for monitoring high-level programs because it requires a means of converting low-level data to an understandable format. However, how this is achieved is not discussed in their project.

A monitored system is viewed by the TMPs as a layered hierarchical abstract structure, but merely in terms of hardware resources; such a structure is unchangeable. Because the monitor does not have the controlling ability to filter data, the behavior of monitoring is not adjustable with respect to what is observed. Therefore, the monitor has no ability autonomously to shift the observation from one layer to another. Thus, the advantage of using abstractions has its limitations.

2.2.5 Monitoring Time Errors by Means of Object Orientation

Tokuda *et al* of Carnegie Mellon University recently presented the architecture of a real-time monitor called ART [21]. The purpose of ART was to solve the special problem in monitoring real-time operating systems, namely time encapsulation. Tokuda *et al* used an object-oriented structure to handle time encapsulation as well as data encapsulation. An expected time-to-reside is defined in an object by a user and is then checked by a method tied to an object. The significance of their work is that it provides a mechanism for dealing with a time-error problem in real-time monitoring and debugging for a distributed real-time system.

The feature of object orientation in ART is limited only in locking the time slice. Overall, ART is not an object-oriented tool since many traits that ought to pertain to an object-oriented model are not available in ART. No efforts were made for abstractly representing a monitored system at the time their paper was published, although ART

developers have since realized the importance of visualizing the system activity at an arbitrary level of abstraction.

No controlling facilities are implemented in ART. The efficiency of data collection is not satisfactory with respect to issues emphasized in this research. No controlling knowledge is applied to run-time monitoring, so that selective data collection can not be performed. Flexibility is restricted because, in working with ART, monitoring is restricted to watching only for predefined types of observation. As a result, it is difficult to produce the type of monitoring necessary for application-oriented programs.

2.2.6 Monitoring with Separation of Collecting and Analyzing

Joyce *et al* at the University of Calgary developed a monitoring system within a distributed programming environment named Jade [23]. This approach is similar to the ideas of the designers of Monit. The difference is that Jade offers two types of trace files. The textual files are more useful for tracking down the cause of an error, whereas the animated graphical trace provides greater insight into the system's overall operation. Joyce *et al* believe that the animated, graphical state displays provide an effective form of dynamic documentation.

The Jade monitoring system was designed to be extendable. The extensibility was achieved by separating the tasks of detecting and collecting information from the tasks of analyzing and displaying the information. The writer of a new monitoring tool is not concerned with the way in which the monitoring information is collected but with interpreting and presenting information to users. The weakness is the system's inability to adapt; changes to the behavior of the Jade monitoring system can only be made off-line, since the collecting process is isolated from the analyzing process

and the communication between these two parts is achieved manually.

Joyce's approach permits a wide range of monitoring tools to be implemented effectively because the writer of each new tool does not have to become familiar with low-level details of how monitoring information is gathered. It is done by converting from primitive IPC events to a complex pattern of IPC events that correspond to high-level operation in an application. This conversion is beneficial to abstracting complexity while also preventing unnecessary details from becoming too involved. Nevertheless, it lacks a fundamental structure to support the abstraction. In consequence, levels of abstraction are not clear for a user to identify; various details are not available, but are limited to two levels: either high or low. Thus, it may be difficult for a user to see and analyze a complex activity, since such an activity is often carried out by procedures presented at several abstract levels.

The design, debugging, and maintenance of a monitoring system are simplified by the Jade monitoring system since the collection and distribution of monitoring information are done with the same IPC mechanism as is used by application processes. In spite of this, distributing and collecting are not facilitated by any filtering process. As stressed in the second issue in the previous chapter, this results in a large amount of data to be collected and thus further limits the inclusiveness of monitoring.

The separation between collection and analysis supports the development of an integrated set of tools that can work effectively, and also relieves the burden of monitoring expenses because the digestion of collected data is carried out somewhere else. The weakness in relating this capability is that it does not assist in providing enough critical data. This is because no optimal control is available for the monitor selectively to collect information, and no assistance in improving the data collection is possible except through manual intervention.

2.2.7 Monitoring with Care of Asynchrony

This tool, developed by Miller *et al* at the University of California, Berkeley, provided a means of monitoring a distributed computation [19]. This model describes process activities in terms of external and internal events. It separately collects external events and uses the addition of daemon processes to coordinate monitoring activities across boundaries. Initial experience in the use of their tools shows them to be useful for measurement studies and for program debugging as well.

The designers at UC Berkeley understood that the main factors that contribute to the complexity of a distributed program are asynchrony, time and delay. To address weaknesses in traditional performance measurement and debugging tools that do not provide enough information to deal with the problems of a distributed environment, and which do not identify problems in programs caused by asynchronous activity and control delays, the designers wanted their measurement system to be a distributed program structured as a group of cooperating processes that may have to run on different systems.

All measurement facilities are designed to support the three stages of measurement: metering, filtering and analysis. Metering is the extraction of data from an operating system for outside processing. But where to meter is determined by a user. Some smart filters are employed to process the collected data, yet no control is supplied to the front data collection; this leaves the problem of blindly collecting data unresolved. Analysis is the extraction of information remaining from the filtering process. Henceforth, meaningful information can be delivered to a certain extent. This monitoring system paid attention to the issue of adaptability for diverse applications. For example, the analysis routines provide means for interpreting the traces generated through filtering; they give meaning to collected data by summarizing and operating on the collected events. The method that lets users produce their own analysis routines according to their particular needs strengthens flexibility of the monitoring

system.

No mechanisms are available for providing an abstract view of the performance of an entire system. Although the behavior of a monitored system is detected through external events and internal events, the distinction is determined by the monitoring system in terms of physical distribution. This is not useful for classifying logical events into levels since a local event in relationship to the other parts in an application program may still need to go across machines. For example, a local record may actually be stored in a disk being allocated remotely. Therefore, something is required to organize those events and to extend the concept of internal and external events to comprise application-level events.

The controlling knowledge integrated in smart filters is predefined. Therefore, this method is insufficient with respect to the criterion emphasized in the discussion of the fourth issue: the knowledge for guiding and controlling the behavior of monitoring must be equally evolvable and extendable as a monitored system changes.

2.2.8 A Hybrid VLSI Measurement Tool

A hybrid performance-measurement system for MIMD multiprocessors, using software (embedded code) triggers and hardware sampling, was developed by Mink *et al* at the National Institute of Standards and Technology [18]. This monitoring tool introduces a minimal amount of perturbation to the executing program. The perturbation was claimed to be as small as a single memory-write instruction per measurement sample.

The monitoring is accomplished by event trace-measurement sampling and resource utilization sampling. Triggering is the detection of a predefined event during program execution that causes sampling of measurement data. Sampling is the collection and storage of a group of measurement data describing computer operations at and between events. The sample may be event trace data or resource utilization data. Event trace data describes operation of the executing software, such as pro-

gram execution time, execution path, and response time. Resource utilization data describes hardware utilization, for example, cache hit ratios, memory access delays, and bus utilization. While this method works nicely with regard to some aspects, it can collect only ill-structured data in the sense that the collected data may not present a high-level view of system performance and may not be understandable, because data to be sampled are predefined and are likely limited at low levels.

Mink *et al*'s project is successful in terms of the goal they set. However, comparing the criteria from the discussion of the stated four issues, the project is deficient in several areas. The emphasis of their research was addressed to low-level monitoring. This is not suitable to the monitoring of high-level programs. Although, with help from a software facility, efficiency can be improved in the sense that triggering events can be collected under some control, the fact that such a selection is determined at the time the monitor is designed restricts flexibility.

Providing an abstract view of a target was not of interest in their project. The monitoring only collects data based on physical events, which are system-defined events. As argued in the preliminary study of this research, it may heavily influence the efficiency of monitoring. Besides, the performance of monitoring is not changeable since all controlling methods rely on procedural knowledge and are fixed once the monitor is designed, including calculations for sampling intervals and conditions for triggering.

2.2.9 Summary of Survey

To summarize the survey, this section will abstract positive aspects of the above reviewed research projects which ought to be considered as part of the measurements for future models. Then the summary extracts their common weaknesses and argues the necessity for an intelligent approach to be pursued. In general, major achievements of these projects can be categorized into three aspects:

- The technology of minimizing interference caused by run-time monitoring is considerably advanced by most of these projects. There are two basic methods for attaining this improvement. One way is to adopt a hybrid model, that is, to take advantage of the little interference contributed by hardware monitors. With the help of hardware data collectors, the software part of a monitoring system is then used only for necessary measuring and for increasing flexibility at the level of application-oriented monitoring. Another method is to separate the digestion of collected data from run-time data collection, so that the overhead resulting from running a monitoring system is limited by strict collection. The complexity of monitoring is substantially shifted from the front stage to off-line stages wherein human intervention and complex analytical methods work together to mimic an internal picture of system performance. Such monitoring systems usually provide their users with the ability of interactive controlling.

- Though data filtering is carried out at a later stage in most of these projects, collected data is finally converted to meaningful information for display. Along with lavish use of bit-map terminals, graphic display, such as charts facilitated with colors, has shown that converted data reflecting a complicated performance can easily be understood.

- Assisting in the debugging of distributed programs or systems has also received substantial attention. One such effort is to detect the time error. Among the methods, some use object-oriented data structures to log the time when events occur, whereas others use trace files to record monitored data with the order of timing. Another method to assist with debugging is to combine monitoring facilities with other facilities so that monitoring, debugging, modeling, analyzing, and tuning can work in a cooperative manner [4].

Despite these great successes, the above reviewed monitoring systems have left room for improvement. Measured against the previously stated four issues, the following common weaknesses exist in these monitoring projects in varying degrees:

- Lack of on-line control over the data collection is a critical weakness. Because of this, efficiency is gained at the cost of reducing the scope of monitoring without ascertaining whether the missed part is critical to the performance analysis. Without the help of inherent knowledge, the complexity of computer systems either proportionally brings complexity to a monitoring tool or shifts to relying on the expertise provided through human intervention.

- The capability of exhibiting a monitored system is still quite limited. In general, only the linear type of information is observed since monitoring basically focuses on individual physical resources and is not concerned with the meaning of monitored data. In consequence, inclusive observation of a complex target is almost impossible. The massive amount of collected data may only reflect the performance of a few physical resources, while data generated from other parts are left unknown. This, then, requires extensive knowledge from an analyst in order to figure out the logical implication from linear-type data to the performance of a computer activity that is usually programmed in a high-level language. Because a modern computer always carries its tasks at several levels asynchronously, this monitoring method cannot collect data in a way which expresses such features; the analysis in terms of data supplied through this monitoring method will hardly remain accurate. Some of these monitoring systems have considered certain kinds of abstract representation, but the abstraction is built in a predefined fashion and only refers to hardware resources. As a result, the efficiency that can be achieved by observing abstract layers of a monitored system is considerably diminished. Some of these systems isolate complexity by considering external and internal events, or low-level events and high-level

events; nevertheless, due to the lack of a structural basis, these monitors have little ability to help organize events into a structural abstraction so that the view of various details in a monitored system can be presented.

- The static feature remains serious, although efforts made in these projects have substantially promoted the degree of flexibility. The most troublesome characteristic is that the monitoring behavior is not adaptable to most of these systems in the sense that these monitors have no ability to adapt autonomously. The fact that the knowledge applied to monitoring is not improved through a monitoring system itself is indeed regretful because the monitoring system should utilize the feedback from monitored data for a finer performance or for adjustment in response to change occurring in a monitored system. Additionally, most functions applied in these monitoring systems are fixed at the time these systems are developed. The consequences are these: abstraction is built only upon fixed hardware layers; types of monitored data, the amount of data to be collected, the resolution of monitoring, semantic concerns about a target, and levels of details are all generally not changeable by a monitoring system itself.

More importantly, these projects have not departed from the conventional methodology which considers monitoring tools consisting of predefined procedures. Unfortunately, this methodology makes it hard to cope with the difficult situation in monitoring modern computer systems.

2.3 Role of Declarative Knowledge

To respond to the challenges brought by newly arisen characteristics of monitored systems, this research asserts that intelligent monitoring is in demand, and that the role of declarative knowledge in monitoring should be greatly enhanced. The necessity

of strengthening the use of declarative knowledge in a monitoring system to a large degree can be generalized into three aspects as follows:

- Monitoring a complex system invokes on-line extensive knowledge to solve problems at the level of a human expert, or quite often, a group of human experts. Additionally, relying on off-line human analysis may result in these problems: anomalies are often overlooked, since representing an error environment is not easy; collected data may not be significant, while significant data are missed; the difficulties in analyzing collected data may become extremely serious.

- Monitoring environments are increasingly diverse, unpredictable, and rapidly changing. A monitoring system is required to have quick response time which exceeds human abilities. In particular, in the case of monitoring a highly dynamic system, an anomaly may trigger a sequence of abnormal symptoms. This in turn demands a monitor capable of quickly detecting changes and promptly switching to those targets at which consequent anomalies may be generated.

- Many activities of a monitored system are imperceptible, so that merely relying on the human aspect for optimal monitoring appears unrealistic. In a process of optimizing system performance, it is desirable to sense minute changes that may suddenly occur and quickly disappear, or may just occasionally happen. With the approach of blind and narrow data collection plus manual analysis based on such data, the chance to discover tiny or occasional variations is slim. Therefore, it is exceedingly desirable to let an intelligent monitor at least partially take over the responsibility.

In brief, the discussion in this chapter indicates that intelligent monitoring supported with declarative knowledge is superior to existing monitoring methodologies which are fundamentally procedure-oriented. The exploration for an approach to intelligent monitoring deserves much attention.

Chapter 3

A Model for Intelligent Monitoring

The preliminary study of this research arrives at the conclusion that intelligent monitoring can be a dominant approach. This chapter designs a monitoring model according to the proposed approach. The resulting model is named AIM, which stands for an *A*pproach to *I*ntelligent *M*onitoring. Since a comprehensive scenario for intelligent monitoring is neither possible nor desirable because it may render the research unfocused, the design concentrates on those aspects closely relating to the addressed issues while giving less attention to other aspects that only remotely relate to this research.

3.1 The Monitoring Environment

Prior to the discussion of the design, a typical environment in which a monitored system is situated should be noted first, so that discussions of restrictions and distinctions imposed on an intelligent system for monitoring can be conducted relatively

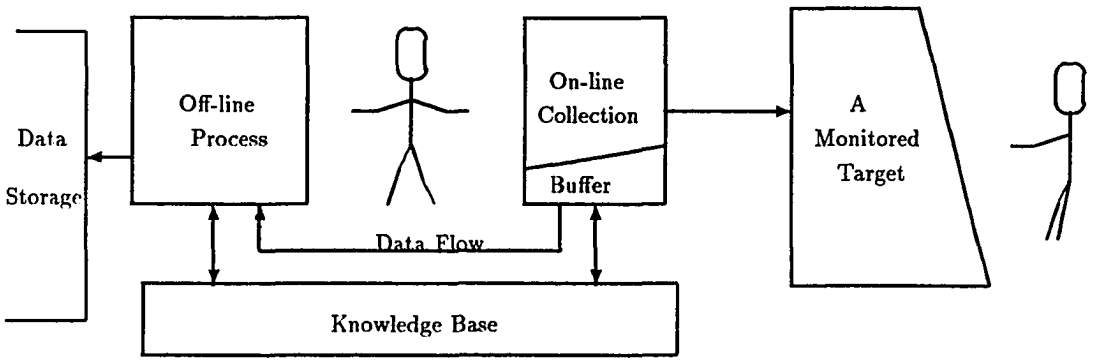


Figure 3.1: The anticipated monitoring environment

easily. Figure 3.1 depicts a general configuration between a monitor and other parties with which the monitor needs to deal.

In this monitoring environment two types of users exist: those who utilize a monitoring tool to observe performance of computers and those who use computers for computer applications. In some instances, they may be the same people, since computer users are frequently concerned with performance of their programs. A monitoring system is intended to benefit mostly the former by providing meaningful data that characterizes performance of computer systems.

The fact that a monitor has to share resources with a monitored system places restrictions on the monitor's use of resources which are chiefly computation and storage. As implied in Figure 3.1, one effort to alleviate the restrictions is to carefully distribute the functions of a monitoring system. The allocation of monitoring mechanisms is principally affected by their needful processing speeds and by the amount of resources they need to use. The resources also include those used in communication at times when part of a monitor has to function remotely. These concerns become more critical in developing an intelligent monitor, since applying intelligence likely requires more computation and space. A mandatory supplement for intelligent monitoring is the use of a knowledge base. For reasons similar to those mentioned above, the knowledge base may have to stay off-line while the buffering mechanism may be

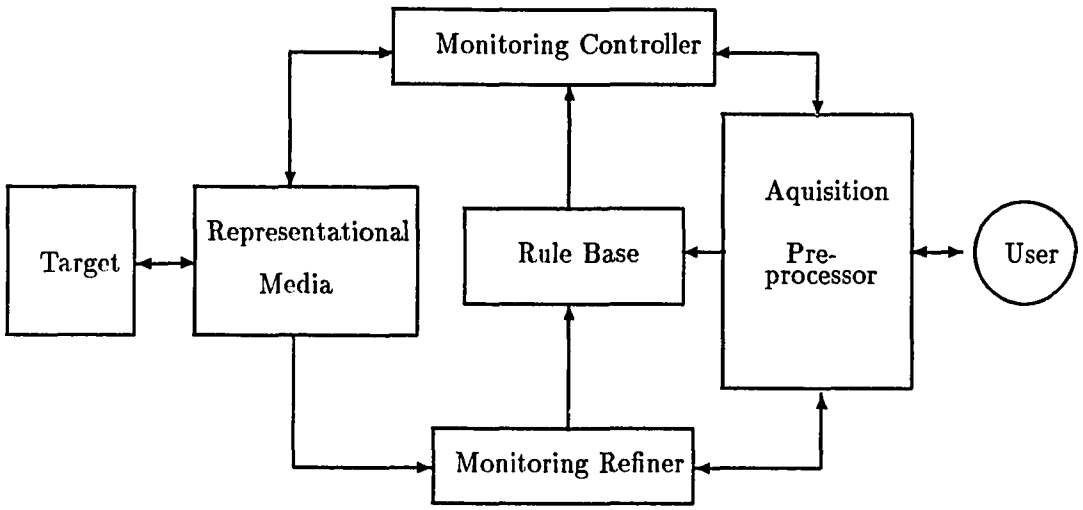


Figure 3.2: An overview of the proposed general structure

needed for on-line reasoning.

Given the general outline concerning the monitoring environment, the remaining discussion is devoted to the design of an intelligent monitoring model with respect to its peculiar environment.

3.2 Overview of the Model

Five components for constructing a framework of intelligent monitoring are shown in Figure 3.2. How these components work together to support intelligent monitoring is briefly introduced as follows:

- The **Representational Media** is responsible for generating abstract representations of a monitored target. The media should dynamically organize abstract computing units into a layered abstraction which represents a target. As computing activities are carried out simultaneously, each abstract entity that features a computing activity should be functionally independent. The quality of the representing media is measured by comprehension and abstraction in resulting representational models.

- The **Monitoring Controller** controls data collection through its reasoning mechanisms. The controller is necessary for on-line filtering. However, its reasoning mechanisms are seriously limited in using resources but are strictly required in obtaining on-line speed.

- The **Rule Refiner** is needed for strengthening the capability of dealing with dynamic features of a monitored system. In general, the rule refiner gets feedback from the collected data and, if needed, refines rules that intimately affect properties of knowledge involved in monitoring.

- The **Rule Base**, as in many intelligent systems, stores rules used by the mechanisms that apply intelligence. One important aspect is the distribution of the knowledge accumulated in the knowledge base. The limitation on overhead may substantially bring about difficulties to the operation of such a knowledge base.

- The **Acquisition Preprocessor** provides users with an interface for passing on their knowledge to other parts of the monitor and for displaying collected data to users.

The relationships of these five components are indicated with arrowed lines as shown in the figure. The knowledge base is conceived by both human intervention and the rule refiner, while the monitoring controller only uses the knowledge base. The possible contribution from the rule refiner to the maintenance of the invoked knowledge has the key influence on intelligent monitoring. The representational media is supervised by the monitoring controller that, in turn, relies on the declarative knowledge available in the knowledge base.

3.3 Desired Capabilities and Distinctions

In thinking about the mechanisms for intelligent monitoring, the first concern is how such monitoring should behave with respect to its particular role. To make this point clear, two general aspects—desired capabilities and distinctions—are discussed in the two subsections that follow.

3.3.1 Desired Capabilities

First, regarding representational media, pertinent criteria [11, 46] can be described in three parts:

- The resulting representation should reduce complexity so that it may ease the understanding of a monitored system.
- The structure of such a representation should maintain the correct relationship regarding time and space.
- The media should be able to reconfigure the representation during on-line monitoring to support data collection in accordance with various concerns.

However, constructing mapping relationships between two adjacent levels of abstraction is an ill-structured task. An often adopted approach to mapping one knowledge entity to another knowledge entity is to apply a set of axioms; this method, unfortunately, is either too restrictive or totally ineffective when it encounters a complex activity [48, 27, 72, 80]. Some effective methods need to be explored for the proposed representational effect.

The intelligence functions are expected to apply two types of intelligence to monitoring: one directs the controlling of data collection, while the other assists in adapting the monitoring behavior. The following capabilities are considered worthy of attention:

- The inference engine should show meaningful data while minimizing the interference due to sharing resources with a monitored system. This amounts to saying that if there is no sufficient time or space to reach a final solution, the inference engine should be able to adjust the strategy of collection and to give a meaningful result; though it may not be precise, it needs to be consistent towards a possible solution, should one exist.

- The intelligent facility should be able to guide monitoring to observe a target at an abstract level wherein a search space can be reduced. In other words, the intelligence should help reduce the cost whenever such a reduction does not harm the quality of monitoring.

- The intelligence should assist in guiding the monitoring at the parts of a monitored system which are obvious sources of faults.

- The refining functions should be capable of organizing knowledge in a way that benefits the application of declarative knowledge in on-line data collection with respect to both accuracy and efficiency.

- The refining capability should guide monitoring during operation, and be able to adjust to changes in a monitored system without abruptly disturbing the monitoring behavior. This requires the refining process to recognize changing patterns in a monitored system, so that uncertain phenomena could be detected while the normal evolution could possibly be accepted.

3.3.2 Distinctions

An intelligent system, in general, consists of two main components—a knowledge base and a reasoning engine [31]. An intelligent system used for monitoring should principally have these two parts. Yet, there are some differences that ought to be

noted. The following are important distinctions:

- The response time is the critical measurement of an intelligent system for monitoring. In normal cases, an intelligent system for general purposes may take a reasonable amount of time to arrive at a solution; for example, one or two minutes may be reasonable. But it is usually not permissible for an on-line system to function at such a speed.

- Flexibility is a dominant factor affecting the quality of intelligent monitoring. One should expect a monitor to function in an actively evolving environment [24, 73]. Thus, the ability to represent a measured system accordingly is crucial.

- Overhead caused by running a monitor must be carefully controlled. Otherwise, the value of adopting an intelligent system is largely disregarded. Unfortunately, this control restricts computing time, memory, and other resources that are used by an intelligent monitor.

In summary, the above distinctive traits relate to three factors of an intelligent system, namely speed, flexibility, and cost. It is unfortunate that these factors are contrary to one another.

While the above general discussion clearly outlines what an intelligent monitor should look like, what is yet to be discovered is how this is accomplished. In particular, what mechanisms should be integrated into such an intelligent monitor? Bearing in mind the stated focus of this research, the rest of the chapter is organized into three sections dedicated to the investigation of mechanisms for the first three components listed above. The design of the rule base and the acquisition preprocessor is given less attention and discussed only when describing related functions in the three emphasized components.

3.4 Representing Media

As more sophisticated computer systems are now readily available, effectively representing complex computer systems becomes more challenging. The fundamental step towards an effective representation is to discover supportive media for representation.

With regard to previous criteria, the technology of object orientation [56] seems attractive because potentially it may bring a number of advantages to intelligent monitoring. For the sake of completeness, a brief study of object orientation is given in Appendix B showing the understanding of this subject relevant to this research. However, conventional object-oriented models, which are often seen as a combination of declarative and procedural approaches of knowledge representation, are still excessively procedure-oriented with regard to the need for representing a highly dynamic system. In particular, the desired flexibility is limited by two static features, namely predefined object classes and hard-wired-in hierarchies. The reasons for this assertion are as follows:

- First, knowledge in the real world evolves over time; objects used to represent knowledge should then be made adaptable.

- Secondly, one area of knowledge is often derived from and composed of other areas of knowledge. This demands, perhaps most importantly, that the structure for representing relationships and combinations of individual knowledge units should be equally adaptable.

Hence, this design investigates a representational framework through introducing an extension to a conventional object-oriented model. Additionally, the representational model should also be supportive in facilitating reasoning mechanisms to fit into the peculiar criteria.

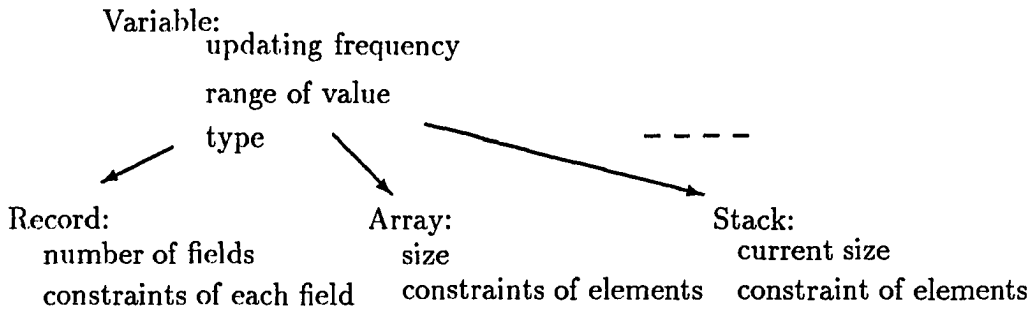


Figure 3.3: Hierarchy of constraint objects

3.4.1 Constraint Objects

Constraint objects are considered primitive entities in representing computing activities. Each constraint is associated with rules and symbolizes a solution which is derived from a rule or a set of rules. An example of using constraint objects to describe some computing entities can be seen in Figure 3.3. The constraint object consists of two attributes: one describes the expected service time and the other describes the range of values. By having more attributes added, it becomes capable of characterizing more specific computing units, such as an array, a record, a stack, and even a function.

The superiority of using constraint objects can be understood as follows:

- Constraint objects provide a reasoning bed at an arbitrarily abstract level. It facilitates the abstraction of knowledge and a knowledge structure.
- The use of constraint objects makes complex constraints manageable. This is possible because a group of constraints may be organized to constitute a macro concept or to reach better granularity.
- Knowledge representation based on constraint objects allows the main computation to take place inside an object, and then conducts the intelligence control in an encapsulated fashion.

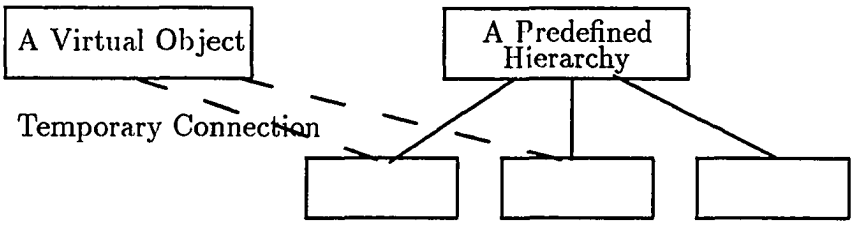


Figure 3.4: An instance of virtual objects

It is noteworthy that, through the use of constraint objects, it is possible to capture integrity constraints from the observed data more easily than from relational data models. Special functions can be tagged on to a class of objects so that data integrity can be maintained. By contrast, in a relational data model, integrity constraints can be specified only at a very elementary level. Conceptual constraints at the information level in a complex structure are hard to depict with a relational model, whereas they are naturally implementable with an object representation. A constraint object may be integrated with attributes such as equations, expressions, variables, and constants.

3.4.2 Virtual Objects

As pointed out earlier, the conventional methodology that creates objects in terms of predefined object classes and binds them in hard-wired hierarchies is too restrictive to represent highly diverse computing activities. For this reason, in addition to primitive constraint objects, the concept of virtual objects is introduced in this design. Figure 3.4 shows an instance of virtual objects that may be dynamically generated in terms of specifications defined in a knowledge base. A virtual object may serve as a temporary associator, linking individual observations in order to constitute a global view; this fits into a situation in which a complex activity is composed of a set of subfunctions for the following reasons:

- A complex computation may lead to the need for in-depth monitoring. Focusing on a subset of monitored activities may help avoid excessive overhead, while

monitoring may still retain the quality of performance if the subset is properly selected and organized.

- Monitoring may be required to provide the information-type description and even possibly the knowledge-type description of a system it monitors. This calls for a means of coordinating and binding related data collections to retrieve the embedded syntax and semantics. A case in point is the monitoring of a query process. A query may need to access several files that may be stored at different sites; each access may be observed independently and simultaneously. Whether the query is correctly processed depends not only on individual accesses but also their order. Associating individual observations may help detect the external logical errors and store related observation in a clustering manner which may substantially ease the analysis of observed data at a later stage.

Hence, a virtual object may be allocated to associate a number of observations in order to provide a comprehensive view. Interestingly, the association of these individual objects exists only during the time their temporary parent—a virtual object—exists. In comparison to objects in a conventional object-oriented model, a virtual object is designed to possess the following differences:

- A virtual object may not follow any predefined object classes and is dynamically generated based on declarative knowledge.
- Upon its generation, it may not need to spawn its child objects, so that some of its child objects already exist and may have been bound to other objects.
- When the attributes and methods of a virtual object are shared by its associated objects, those attributes and methods may not be passed to the classes to which associated objects belong.
- A virtual object may have neither a parent object nor its own methods. It is seen by users as if it functions with no difference from a conventional object.

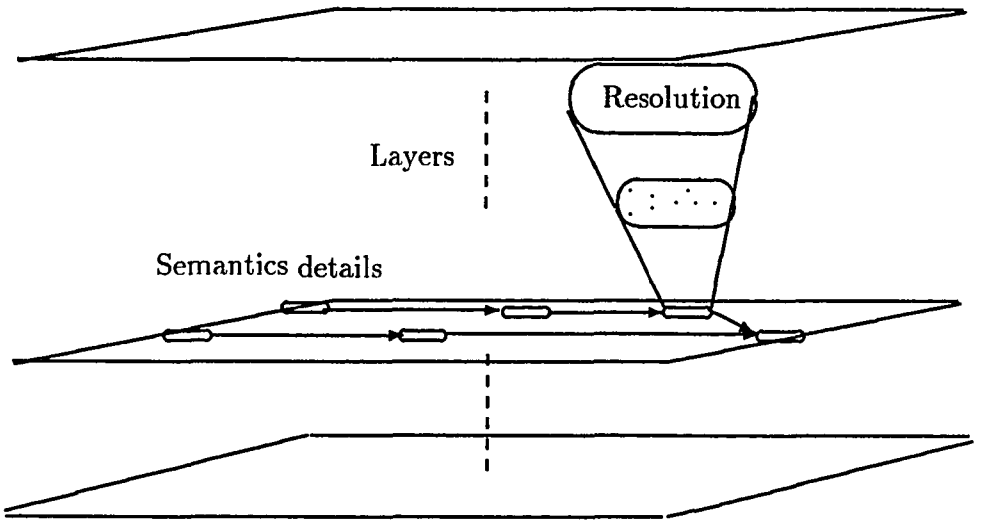


Figure 3.5: A highly abstract computer architecture

- A virtual object has no right to terminate the existence of associated objects, but it may prevent termination of associated objects.

The concept of virtual objects is somewhat related to the concept of delegation [47, 71, 70] but with a more coherent structure. In other words, one may think that a virtual object receives data from other objects and displays such data in light of its own constraints. A virtual object is defined by rules, so that it may vary with applications. While the formal specification of a virtual object will be discussed in the next chapter, the investigation of representing monitored targets continues with the use of the above proposed objects.

3.4.3 Layers, Details, and Resolutions

Constraint objects are one type of complexity-hidden structures for abstracting knowledge entities. With means of both primitive and virtual objects, the abstract representations of a monitored system are developed by considering abstractions in layers, details, and resolutions as shown in Figure 3.5. This means to divide a complex system into a layered structure first; at each layer, monitoring focuses on a subset of

computing units and their relationship; for each computing unit, observation further focuses on concerned attributes and describes these attributes with different resolutions.

Abstract Layers

Two critical issues must be resolved first in order to effectively support a layered abstraction:

1. The behavioral description at an abstract level, possibly including analytical knowledge, must be properly integrated.

2. Consistency between the lower-level abstraction and the higher-level abstraction must be maintained.

An abstraction possibly satisfying the above criteria may be achieved by means of mapping from an abstract system to a more concrete system which becomes available due to the use of virtual objects as detailed below.

A complex computing system should be first decomposed into a layered structure in which each layer covers the complexity underneath it and virtually provides users with a more powerful set of functions in comparison to its immediately lower layer. Thus, such layered structures prevent users from getting into untrackable lower-level activities. Figure 3.7 shows a layered abstract structure resulting from decomposing a computing architecture that people often confront. Monitoring based on this abstract architecture can further be benefited by associating observations at a layer so that abstract connections between layers can be managed. Figure 3.6 shows that a virtual object is created to associate a subset of functions at one layer and to connect the observation at the higher layer. Such a virtual object, which could be a primitive constraint object predefined as a function type or a record type, is actually the substitution of a monitoring object at the higher layer. With such a layered abstraction,

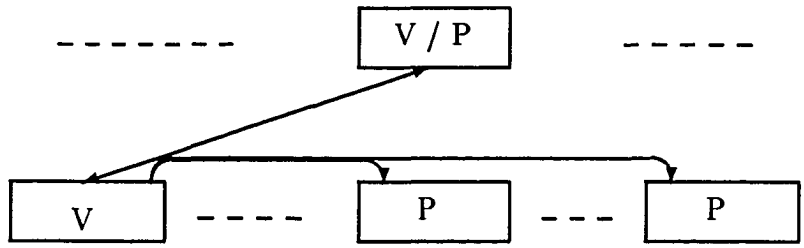


Figure 3.6: A virtual object for associating individual observations

Level 1: a machine with numerous processors, sufficient storage, and dedicated auxiliary devices; all resources seem to be nearby to users

Level 2: multiple user processes

Level 3: virtual I/O devices

Level 4: distribution of resources

Level 5: hierarchical memory

Level 6: virtual CPUs

Level 7: core machine

Figure 3.7: A layered abstract structure

computing activities in a complex system are considered to be carried out by one of the alternate virtual machines. This type of abstraction is beneficial, particularly to the group of application-programming users, since the understanding of a system can proceed with the top-down approach and often does not have to consider those activities at low levels. The layered representation fits into the needs for observing high-level programs. Figure 3.8 shows that a query request may involve a series of actions; one is to retrieve a number of files storing related data. The access to each file is converted to retrieving a number of logical records. Each logical record is mapped to several physical records; producing a logical record involves having mapped physical

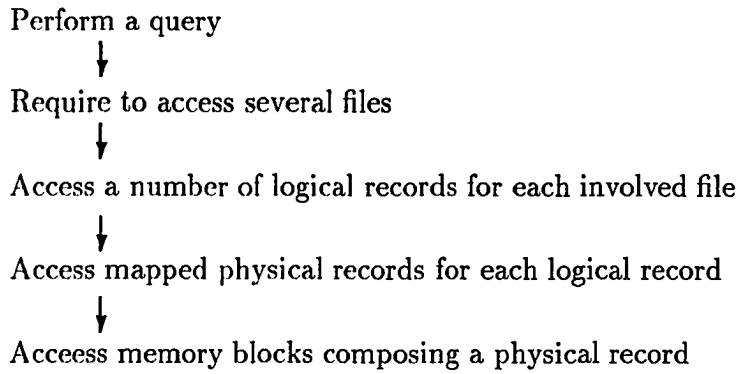


Figure 3.8: A layered abstract structure in processing a query

records retrieved. This again calls on a set of I/O operations for the retrieval of physical records. Whether the data is stored at different sites or in different storages within one site is observable only when a user watches the process at low layers. A mapping between Figures 3.7 and 3.8 can be seen through such understanding: a query process is issued from one of the user processes; each user process runs in a virtual terminal that may be connected to a remote machine; retrieving the files involved in the query process may require visits to different sites of a computer system; a file and perhaps even a logical record may be stored in different types of memory; moving blocks of data is performed during assigned CPU slots and which CPUs actually perform the access is hidden from the file system as well as the I/O system. So, a query process is in fact carried out at all levels as outlined in Figure 3.7. The lower-level activities beneath an abstract layer are only of concern if a more detailed analysis is needed. An exhaustive track-down is seldom necessary and usually not practical.

Monitoring based on the above-described layers of abstraction can be managed by allocating objects, both primitive and virtual objects. While a set of primitive objects physically collects data, virtual objects may associate and coordinate the data collection performed by those primitive objects. A few things need to be done in order to prevent such a multi-layer observation from significantly interfering with

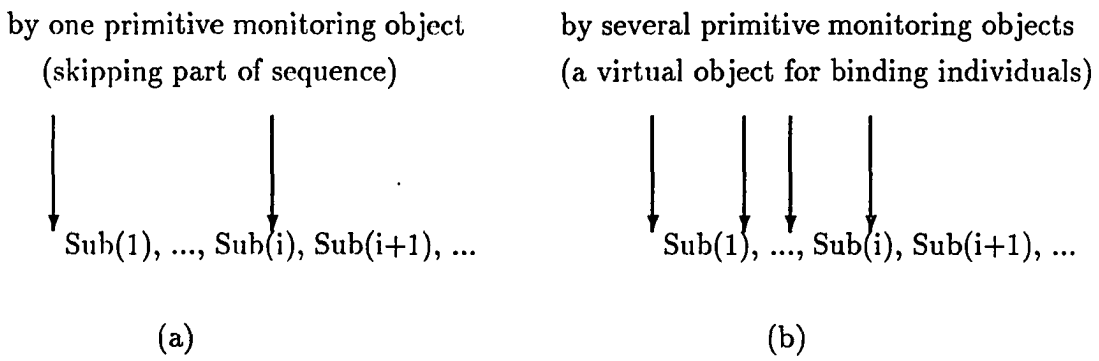


Figure 3.9: Abstracting computing activities at a selected layer

the monitored system:

- Primitive monitoring objects should be utilized multiply by virtual objects; data collected by an object should be utilized in composite monitoring as much as possible.

- While monitoring at a certain layer is not performing, the corresponding virtual object that maintains the abstract monitoring structure should be suspended and its needed child objects be released.

- At the each layer, monitoring is again deliberately abstracted so that a sequence of subactivities may be monitored through the strategies depicted in Figure 3.9. Two ways are described in the figure; one of these is to let one primitive object cross several subactivities, and the other is to selectively monitor part of the sequence.

Additionally, the overhead can further be reduced by focusing on emphasized semantics and attributes as well as minimizing the degree of resolution and details within a selected abstract layer (which are to be discussed in the sections that follow). Hence, the overhead may hardly be more than the linear-type data collection.

Semantic Details

At a selected abstract layer, the next process is to construct constraints that describe the features of a target viewed from this layer. This is carried out by first projecting semantic details and then structuring the resolution of the description. To avoid any confusion, the need for further abstraction and difference between resolution and details in this context should be clarified at this point. The details mean a number of semantic aspects of a monitored activity viewed at a selected layer. For each attribute to be observed, the next concern is the degrees of making monitored attributes distinguishable, which is the issue of an appropriate resolution.

For example, monitoring a query at the user interface level or at the system level provides data for different groups of users. This is required because an end user's concern is usually different from a system analyst's, and a database user may have distinct interests opposite those of the database designers. At a selected level, the semantic details of monitoring should further vary, depending on how many semantic attributes are to be seen in individual cases. It is not always good to monitor as many attributes as possible, since more comprehensive details are balanced by having to stay at a more abstract layer and/or with a lower resolution; this is true because monitoring is limited by computation and the use of storage. For selected attributes, the amount of detail which a monitor is required to distinguish becomes the issue of the degree of resolution. The resolution of monitoring certain features varies according to the context. It is generally suggested that excessively high resolution be avoided whenever data collection with a lower resolution may characterize observed attributes.

An object in the real world consists of many attributes, so observing all of them at the same time may not only cause substantial overhead but also bring difficulty to the analysis of collected data. Hence, projecting computing units into certain semantics-emphasized abstract units is a further step in achieving abstraction. It may also require the abstract representation of a computer system to cohere with

multiple semantic descriptions [44].

Resources constructing a tangible system may roughly be categorized into two types: hardware resources and software resources [28, 65]. In addition, the ways of utilizing resources to generate expected outputs are expressed through logical steps which are the intangible part of a system. Typical examples of intangible resources are policies for queuing, accessing, sharing, and others, as well as algorithms that execute policies of data manipulation. Furthermore, the above monitoring may be restricted within certain parts or areas.

To summarize the above discussion is to say that monitoring activities regarding semantic concerns may be characterized in the following ways. First, they address two basic types of data collection—data flow sampling and event-driven collection:

- Sampling data flows is often preferred by an analyst who observes asynchronous activities and changes with unknown patterns.
- Recording event-related data is preferable in cases concerning the logical structure of a monitored activity.

Secondly, based on the above types of data collection, semantic concerns of monitoring contain the following anticipations:

- Resource-oriented monitoring. This type of monitoring mainly provides the size, position, distribution, and utilization of each resource. This type of information is significant, especially when a system is being tuned.
- Logic-oriented monitoring. This type of monitoring focuses on a set of components involved for a logical performance. An example of this is the case of observing the execution of a policy, in which a set of participating computing units may simultaneously be monitored.
- Geography-oriented monitoring. The orientation of this monitoring intends to observe components which are physically related. It is preferred when one

confronts a complex target. Globally speaking, such monitoring may watch for how associated processors and I/O channels within certain parts of a system coordinate, or how computing activities within a network station perform.

The above types of monitoring may deliver different types of information with different costs. Moreover, these kinds of observations can be executed in an integrated manner, and they may be performed through one composite monitoring object.

Finally, as an alternative, another approach to indicating the attention of monitoring can be in terms of exhaustible and nonexhaustible resources. For example, memory, processes, and I/O channels are exhaustible resources; whereas algorithms, policies, and functions are nonexhaustible.

Resolution of Description

Once semantic features to be monitored are selected, the degree of descriptive details needs to be considered. The resolution of monitoring can be defined in terms of physical resolution and logical resolution. The physical resolution relates the types of data used for recoding the observation and the amount of space is needed to hold the data. The decision made about the degree of resolution depends on how significant the data is and how critical the monitored activity is.

In monitoring the service time of a job, a monitor may give three types of data: a Boolean number representing something like the fact that the job gets faster service or the job has a higher priority for service; a multi-value description—such as the low, medium, and high speed of service; a direct reading from a timestamp. Along with a higher degree of resolution, more computing time is needed to record the observed data and more room is needed to store the reading. Hence, types and perspectives of data being collected affect the efficiency, while none of them may necessarily hurt the correctness of monitoring. As handled in this example, one may use different expressional media to describe facts at related levels of abstraction for

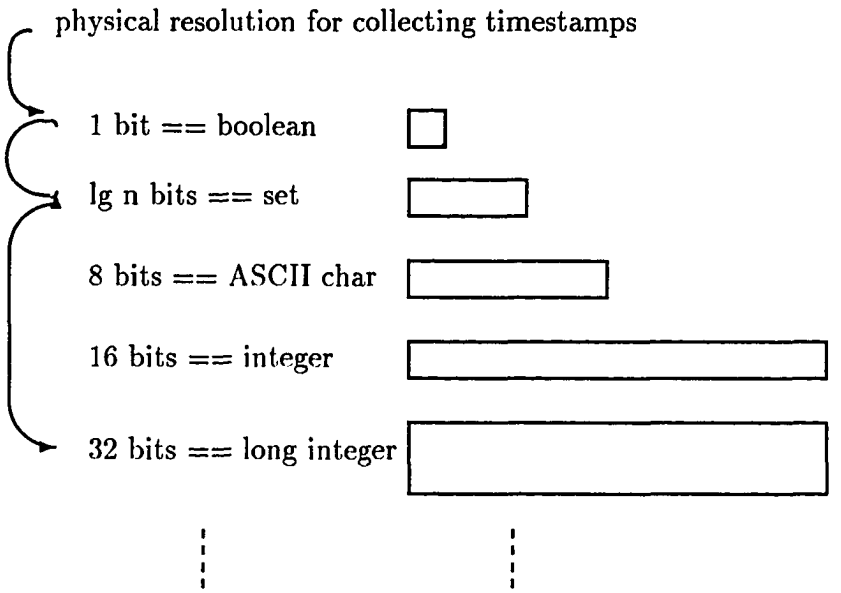


Figure 3.10: The structure for changing physical resolution

a target. Boolean numbers are used for the most abstract descriptions; discrete sets are used for the moderately abstract descriptions; a long integer is used to directly record the timestamp. The preference in the degree of resolution distinctly affects the comprehension of data collection. Achieving both an appropriate resolution and more comprehensive coverage is often more desirable than an exhaustive but narrow observation. A linked structure supporting the change of the recording resolution during on-line data collection is illustrated in Figure 3.10.

Based on physical resolution, logical resolution concerns the granularity of data collection. Depending on the amount of information gathered for monitored features, descriptions of logical resolution may be categorized into three levels: data, information, and knowledge. The categories can roughly fit into a diagram of knowledge evolution which is shown in Figure 3.11. Each level of description, however, requires varying efforts with regard to computation and storage. The data-level description is the most economical description. Examples of the data-level description are binary indicators and counters; they can be held by specific hardware registers as well as

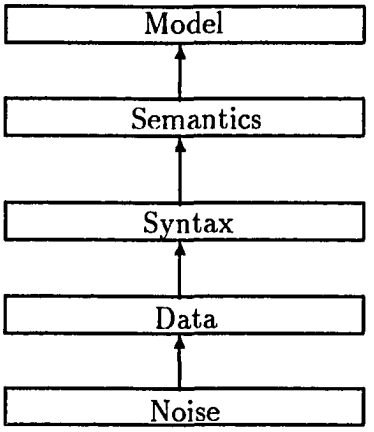


Figure 3.11: Levels of logical resolution

software registers. Binary indicators can denote the availability (or unavailability) of a resource and the running (or blocking) state of a process and the idle (or busy) state of a resource. Counters may be used in denoting the degree of multiprogramming, the length of a resource queue, the duration of a service burst, and others.

The information-level description collects data with respect to the embedded syntax. In monitoring the performance of a statement in a high-level program, primitive elements such as variables and constants invoked in the statement may be collected in a relevant manner. Moreover, the structure used to store the collected information is also slightly more complicated due to the need for associating indices to maintain the collected syntax. In observing a function call, the number and the order of parameters ought to be recorded in addition to the contents of those parameters, and all recorded data should be associated.

The knowledge-level description may concern even higher granularity of data collection and be intended to collect the semantics of monitored activities. Continuing the previously outlined observation on a function call, the collection at this level might need to record values of actual parameters as well as values of returned parameters in a relevant manner. Furthermore, the storing of recorded information might need to distinguish between fed-in values and returned values, in addition to correctly pairing

those records.

More descriptive than all these three levels of descriptions, the model-level description can be derived by allocating a variety of counters for statistical numbers. Basic analytical numbers such as the balance and variance of monitored data can be easily yielded from mean values such as the prospects about the number of accesses to each file type, the number of high-level language statements of each type executed, and the number and types of interactive commands processed. Nevertheless, the derivation of models usually involves a high cost in maintaining the large amount of detailed information in order to finally produce knowledge; such description should be provided through an off-line analysis.

A more detailed example may help consolidate ideas about varying resolution. Thus, the case of monitoring a division between two integers is examined hereafter. There may possibly be different sets of constraints defined to collect significant data for a subset of attributes of this computation. One possible set of constraints for monitoring this computation is instanced below:

- Syntax:

  the format of the division statement is $z = x/y$, where z must be a variable and downward compatible to x and y.

- Semantics:

  IF y is updated between the two readings on x and y, the access is not valid.

- Input/Output:

  the absolute value of z is less than x; if x and y have the same sign, then z is positive; otherwise, z is negative.

- Time:

  the tolerable time for completing the calculation is $mean \pm \delta$.

If the above constraints were applied when the calculation was monitored, and nothing was collected, this means that the monitored activity had performed as expected in relevance to the resolution at this layer. There might possibly be some non-trivial anomalies existing. However, as far as this abstract level was concerned, the calculation would be considered normal. If any constraints were violated, the related phenomena could be collected in different ways: for example, by a *true* or *false* indication, by classifying measures such as *very poor, poor, normal, good*, and *very good*, or by a direct recording. With more restrictive constraints, the observation may collect more information that is not visible with the original set of constraints. For example, additional constraints such as the following could be added to the previous set:

- Time:

  time to retrieve x and y is *mean retrieve* $\pm$ δ_1;

  time for calculation is *mean calculation* $\pm$ δ_2;

  time for assigning the result to z is *mean assign* $\pm$ δ_3.

- Input/Output:

  the assigned value of z should be less than 0.5 different from the value computed.

The second additional constraint may help a user examine whether truncation is appropriately processed by a compiler. With two additional constraints, the monitoring is then able to detect phenomena that may not be seen at the level of abstraction set by the initial set of constraints. Yet, this is still relative. The resulting set of constraints is still insufficient to reach the degree of resolution which a user may like to see. Finally, it is worth mentioning that two types of abstraction are discussed in this section. Whereas one represents logical relationships among high-level operations and low-level operations of monitored activities, the second is less dependent on domain knowledge and further derives levels of abstraction from an abstract layer resulting from the first type of abstraction.

3.5 Mechanisms for Data Collection

Following one of the fundamental principles established for this design—to enhance flexibility—a multi-level controlling methodology is considered for applying intelligence [36]. The intelligent functions can be categorized into three levels:

- Using default logic to control the constraint-based data collection is the first process that directs monitoring to select data purposefully as well as efficiently. Constraints are converted from rules, and rules are defined upon domain knowledge. Constraints are often applied in a group manner.

- Triggering methods are defined to conditionally enforce constraints. Criteria applied to trigger data collection at certain parts of a monitored target during certain periods are basically composed of domain-independent knowledge.

- Refining constraints and rules used for triggering constraints is accomplished through these processes: one compromises exceptions, the next one updates controlling criteria along with the changing of a monitored system, and the last one is for adjusting the strategy of refining.

In particular, these mechanisms are designed to function in a way to maximize the advantages which result from the representational media. The continuing discussion of this section will exhibit the first two types of mechanisms and leave the design of the refining process to the next section.

3.5.1 Using Defaults

Most of the existing methods of conducting reasoning [32, 45, 7] are too expensive as well as too strict to fit into the scenario of intelligent monitoring. In most monitoring situations, a hastily invented solution may be better than a mistaken solution or

a solution requiring heavy computation. The method favored for the purpose of reducing reasoning time is the use of defaults.

The principle underlying the endorsed default reasoning is that if no data is collected when a selected set of constraints is applied, the conclusion is that the observed activity presumably is as good as predicted [30, 5, 62]. The principle may be written in the following form:

$$Sth(x) \overset{\text{no exceptions seen}}{\Longrightarrow} Perf(x) == Perf(c), \quad \text{where } c \text{ is a typical instance in } Sth.$$

It means that since x is in *Sth* as known and no exceptions have been found, it is understood that the performance of x is about the same as the performance of c, a typical instance in *Sth*.

Two default constraints are instituted as follows:

1. *prerequisite*: addition with two floating numbers,
   $justification_1$: a similar machine does this with x seconds,
   $justification_2$: addition with two integers requires y seconds,
   conclusion: upper bound is $x + \epsilon$ seconds and lower bound is y seconds.

2. *prerequisite*: sending a z kilobyte packet in a network during peak hours,
   $justification_1$: a recorded average is x seconds per kilobyte,
   $justification_2$: jobs executed during peak hours are on average $y\%$ slower,
   conclusion: upper bound and lower bound are $(x * z * (1 - y\%) \pm \epsilon)$.

The above ϵ indicates a tolerable error. Values of above x, y, z are left to be specified by each application, so that rules are more widely applicable.

Conditioned Defaults

Default reasoning is tentative in nature. What is gained is that the time spent on reasoning may be much less. In practice, however, defaults need to deal with exceptions, so they need to be organized into a structural manner [76]. Taxonomic

hierarchies appear to be feasible as a structural basis. As pointed out by Reiter [61] and Lukaszewicz [50], in a broad sense, default theories can be naturally regarded as a special class of axiomatic theories in a nonmonotonic logic. This occurs because default reasoning often arises from prototypical situations. The default that says

$$\frac{IntegerAdd(x):\ \ FasterThanFloat(x)}{FasterThanFloat(x)}$$

means that most times of integer addition take less time than a floating addition. This has a pattern similar to the example, given by many authors [62, 16], which says most birds can fly. Nevertheless, there are often fallacies hidden in some default statements. Those fallacies could largely increase uncertainty and perhaps lead to default reasoning nonmonotonic up to an inapplicable degree. What makes the problem so serious is that defaults may be used to reason under any circumstance. To compensate for this weakness, defaults need to be equipped with conditions so that, for example, the above default can be rewritten as:

$$\frac{IntegerAdd(x)ConditionedBy(SameLoad(x)):FasterThanFloat(x)}{FasterThanFloat(x)}$$

which can be understood to signify that most times of integer addition spend less time than a floating addition if they are performed by systems having the same load.

Reasoning with Confidence

Unlike "most" used in conventional default methods, defaults should go along with a degree of confidence. Starting from the default's defaults such as "most," and "often," the defaults may gradually be attached with more precise coefficients, or more specialized defaults may be developed to extend the hierarchy initiated by the original default. The above argument amounts to saying that defaults should be refined in terms of individual cases and should also be generalized to increase the applicability. Heuristics are used to help make the decision of applying defaults

so that the default reasoning may yield more confidence. Heuristic principles—for distinguishing priorities of constraints to be attempted—can be the following:

- Heuristic 1: Most recently used. That is, if a default is used more than others during a recent period time, the default has a higher priority over others in being applied.
- Heuristic 2: Least computational. This is to say that if a default requires less computation time in comparison to others, then the default has a higher priority over others attempted.
- Heuristic 3: Least covered. This criterion guides the trial of application of such defaults in order to involve as few components as possible.
- Heuristic 4: Most anomalous. This criterion guides the monitoring to shift its efforts to the areas appearing to have the most abnormal symptoms.
- Heuristic 5: Oldest. A default may be selected simply because it has existed longer than others.

However, on-line monitoring should not and may not be able to figure out the priority of each default or each set of defaults. Rather, it assumes a default hierarchy is well organized in terms of priorities. One way to accommodate the influence from different heuristics is to give a combined weight to each default, and to link those defaults of the same semantics in an order based on the compound weight.

The above statements suggest that default constraints should be extended to comply with exceptional cases. Heuristics influencing the application of constraints should also need to be improved to respond to changes in a monitored system. This type of function is considered the refining capability of a monitoring system and will be discussed shortly when refining mechanisms are investigated.

Relative Defaults

Defaults may depend on other defaults. As mentioned before, such defaults are capable of being applied because of the use of composite monitoring via virtual objects. The relation between two or more defaults may need to be expressed through some equations. An instance is that a default for one of a sequence of activities is often determined by its precedent activities such as

$$\mathrm{dft}(x_i) = \mathrm{func}(dft(x_1), dft(x_2), \ldots, dft(x_{i-1}))$$

This requires yielding a default at run time in relation to others. The variation resulting from the above function can be one like

$$\mathrm{dft}(x_i) = \max(dft(x_1), dft(x_2), \ldots, dft(x_{i-1})),$$

which may mean that $\mathrm{dft}(x_i)$ is dependent on the maximal one among those defaults applied to previous activities. This relative constraint is suitable for a number of cases, particularly those relating concurrent activities:

- Processing a query needs to wait for retrieving files that are located at different sites.
- Releasing occupied resources will not occur until all needed resources are obtained.
- Sending a message starts only when channels are free, which means that all issued messages are delivered.
- The finishing signal of an external sorting will not be generated until the sorting in each file is finished.

Since the above activities are often carried out in a concurrent computing environment, the sequence is not predictable. In consequence, constraints have to be arrived at dynamically.

The use of relative constraints helps correctly control data collection. For instance, in the case that an entire system runs slowly but smoothly, it is ideal to collect data only from the first activity in the sequence of activities if an anomaly may likely be caused by the first rather than subsequent activities. Such detection is possible because of the composite monitoring with which the influence among individual activities can be sensed. Now, a new issue regarding default reasoning is that of working together with a layered abstract knowledge structure, an issue which will be addressed in the next section.

3.5.2 Reasoning at Abstract Layers

Reasoning with abstract knowledge in an encapsulating manner has been recognized as a mechanism naturally supplementing object orientation. The motivation for using this method is to make reasoning less time-consuming. Working together with defaults, the reasoning procedure can be generalized into the following steps:

1. The solution of a problem is first attempted by ignoring unimportant details and using high-level knowledge abstractions.

2. When a solution to the problem in an abstraction space is found with sufficiently high confidence, all that remains to be done is to account for details that support this hypothesis. This can be accommodated by simply recording indices to those details.

3. If a recognition process fails at some abstract level, (i.e., inconclusive answer), then the reasoning process switches to a lower level wherein more facts are readily available.

With regard to effectiveness, hierarchical reasoning is fundamentally suitable to furnishing on-line control over data collection.

To control the communication between each layer of knowledge sources, module-type reasoning methods may suit the need for interaction between layers. The method may function based on different types of layered abstraction spaces; one is the uniform abstraction space whereas the other one is the heterogeneous abstraction. Uniform abstractions may use the same vocabulary as final solutions and may differ only in the amount of detail. The heterogeneous knowledge space may lead to the necessity for using different vocabulary to describe the knowledge at each layer. The example of reading from a timestamp given in section 3.2 is one such case which uses a heterogeneous knowledge space. Accordingly, the methods for controlling data collection may have to be different at each layer. An intelligent monitor, observing the layered representations of computer activities, may often have to deal with heterogeneous structures of knowledge space, such as virtual memory and physical memory, processes and processors, logical records and physical records, symbolic names and their correspondent identifications, and so on.

3.5.3 Shallow Reasoning

It is considered shallow reasoning to distinguish the significance of data; this method helps provide quick solutions to problems of recognizing types because it embodies only what to do but supplies no information as to why it should be done. During monitoring, most of the time there is no need to know the "whys and wherefores" of the relation between the data observed and the faults ruled, because the duty of monitoring is to provide significant data and leave judgements about what brings up the data and why it causes the phenomenon to be made at a later-stage analysis. For example, in the case of a failure resulting from a file opening action, the fault can be traced to a non-existing file name being given or other possibilities. When the

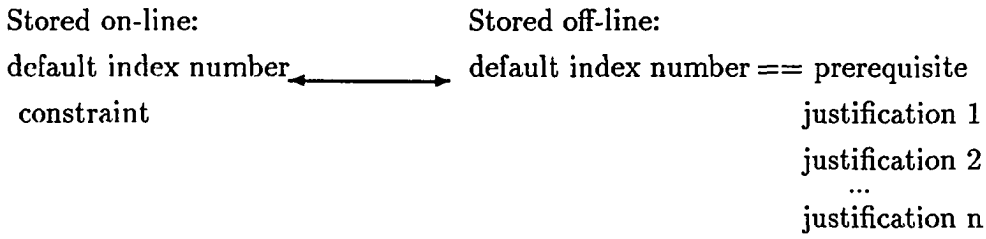


Figure 3.12: Storing default constraints distributively

knowledge is applied, it is not necessary to specify how a file name is connected to the action of opening a file. Such a detailed understanding of underlying reasons that links one phenomenon to others often becomes appealing during an off-line analysis.

Additionally, the knowledge structure—in which rules are represented through constraints, and constraints are organized based on object orientation—suits well shallow reasoning, which is appropriate for reasoning with time limitation [26, 60, 63]. By shallow reasoning, the idea of keeping the on-line monitoring mechanisms as small as necessary can possibly be implemented. Figure 3.12 shows possible conservation of resources by leaving the main part of a default in a remote storage. Since most constraints can be expressed in a numerical manner, the ratio of the room for storing constraints to the one for storing the entire rule structures is very small. Finally, the method of shallow reasoning only works efficiently when it functions with a complexity-isolated model to produce the simple relationship of cause and effect. Involving fewer constraints at each abstract level could facilitate the shallowing reasoning's being carried out.

3.6 Refining Methods

The ability to learn must be part of any system that attempts to demonstrate general intelligence; it is particularly important for an intelligent system designed for monitoring since observed data are available for use as learning resources. However, this does

not mean that learning occurs without being influenced by human experts. Instead, the learning capability pursued in this design is knowledge-rich learning. On-line learning during monitoring and off-line intervention from human experts may work cooperatively to complement each other. The former may be mainly achieved by the inductive learning method while the latter may be achieved through deductive learning. Notice that the expected dependence on human expertise is different from requiring on-line expertise from users.

Refining methods considered in this design are primarily situated in two extensions of the default theory. The first is to introduce compromised taxonomic defaults by extending taxonomic defaults [29], while the second is to apply modeled defaults. While the specification of refining methods will be presented in the next chapter, the general strategy of refining is to be examined in detail after some theoretical thoughts are introduced hereafter.

3.6.1 Underlying Theories for Refining

The refining methods considered are principally found by following an induction approach. Nevertheless, in contrast to many other inductive learning methods that are situated in a knowledge-poor basis [54, 35], the learning process endorsed in this design, more precisely called the refining process, is to have support from a rich knowledge base. The gain is operational while the loss is limited learning capability.

In addition, generalization and specialization [64] are other learning technologies that are of interest to this design; it becomes feasible to integrate them into the learning pattern studied in this design due to the representational media. Generalization and specialization transform descriptions along the set-superset dimension in both directions in a way that makes descriptions more effective, but it may be falsity-preserving because an inductive generalization is used. Thus, aimed at the reliability of an inductive refining, a refining process, which is fired when accumulated statis-

tics reach the refining condition, is designed to organize knowledge units into general classes depending on the degree of maturity of such knowledge entities. The refining process, fired when some accumulated recording reaches the refining conditions, occurs within a class as well as across classes. However, moving those knowledge units across class boundaries will mean more resistance than moving within a class.

3.6.2 Taxonomy with Compromise

Data is conditionally developed into knowledge in a form of defaults, and then defaults comply with hierarchical structures through evolution. By means of compromise, default hierarchies may retain entities that do not logically follow and have various degrees of insufficient confidence in describing truth.

As proposed at the beginning of this chapter, the refining process focuses on three types of knowledge involved in monitoring. Within the first type of evolution, there are at least three possible ways to update existing knowledge. One extends default hierarchies. Triggered constraints are recorded; the number and frequency of triggering are accumulated to possibly fire a refining process for accepting unexpected cases as exceptions. The second is to treat newly-met phenomena as new cases. This is different from the first one regarding their treatment. Generally speaking, the first way is to extend a hierarchy vertically; whereas the second way is to horizontally extend a hierarchy. The third refining function serves to adapt existing defaults. This is necessary since little data being collected may imply that applied defaults are not capable of characterizing a monitored activity. In such a situation, defaults may need to be adjusted to get a sufficient amount of data so that the target can be featured through the improved default set. Although with the third method, a hierarchy does not change regarding size, its precision with a monitored target may be substantially improved; or it may be considered as a way of specializing knowledge. In addition to statistical numbers, some principles ought to be applied to decide whether a refining

process is triggered to evolve existing knowledge. Typical examples may be like these:

- An accepted case must not worsen the stability of an entire system.
- An accepted case must have a regular pattern. This means that if a phenomenon happens frequently but with an uncertain pattern, it may never be accepted.

3.6.3 Modeled Defaults

The second refining strategy, called modeled defaults, is initiated by the default model theory [51]. The principles underlying the default model theory are to minimize the total number of cases and to maximize the coverage of each case. While, at a glance, it seemingly contradicts the refining method above described, it will be shown shortly that the method indeed works nicely with the first method. Along with the extension of default hierarchies, generalization perhaps is a critical medium to withstand the resulting growth for the purpose of keeping knowledge structures operational. Two properties of a knowledge structure are chunking and clustering, which may significantly affect the efficiency of applying knowledge. They in fact serve as guidelines for maintaining the modeled defaults. The techniques for realizing these two properties are discussed in the following two paragraphs, respectively.

Techniques for Chunking

Chunking is a learning mechanism [43] and is useful to this design because it leads to faster and more effective problem solving [9]; the resulting knowledge structure well suits the demands for less computation in accessing knowledge.

Constraints that can distinguish more instances are assigned a high priority for application. One way to achieve chunking is to promote those constraints to a higher level in a constraint hierarchy. This is not equivalent to substituting other constraints, in spite of the fact that it may likely have the same consequence; a constraint that

does not show good efficiency in comparison to others may have less chance of being applied. Efficiency means determining whether or not those constraints are helpful in demonstrating characteristics of a monitored activity. Constraints associated with exceptional cases do not necessarily remain as exceptional knowledge entities. Instead, an exception may be promoted on the way to replacing a normal case which it contradicts, if the exception happens more often and does not bring up resulting abnormal phenomena. The promotion is realized by moving related constraint objects to the position having the highest priority for being applied.

Limiting the size of a default hierarchy is another method for achieving chunking and is an important factor facilitating on-line reasoning. This leads to discarding some knowledge entities such as those cited below:

- Rarely used constraint objects could initially be considered as descriptions for exceptional cases and might eventually be dropped off.
- The influence of some heuristics may be decreased to the point at which those heuristics seemingly do not exist.

Techniques for Clustering

In pursuing flexibility, a constraint object should be discouraged from increasing its size. On the other hand, seeking effectiveness drives the refining process to construct a better clustered hierarchy, so that an opportunity to apply several related objects for a compound observation can be discovered. Also, clustering may be built up with the relationship of multiple semantics, resulting in allowances for multi-parent hierarchies. As a compensation, however, inheritance may not be strongly supported.

Thus, constraint objects are gradually connected (possibly in a partial order) in terms of their characteristics. Guided by heuristics, the structure of objects may result in more specific objects being located at lower levels whereas more generalized objects are set near the top. This avoids the need for developing a large-size constraint object;

it encourages a search for finer results to follow the layered knowledge structure. In other words, knowledge is applied as a set of entities located close enough to be easily accessed. Each entity does not heavily overlap others but, instead, collaborates with others to form a macro concept.

3.7 Summary of the Design

The orientation of this design is to enhance the role of declarative knowledge. To obey peculiar criteria imposed on on-line intelligent monitoring, the design is situated in a more declarative object-oriented model. With this as a basis, the representing media is designed to abstract complex systems by supporting the preference for layers, semantic concerns, and resolution; taking the representational framework as a reasoning bed, the controlling mechanisms apply constraint objects to selectively collect data in terms of the performance of individual activities as well as composite activities; finally, the refining process further classifies invoked knowledge and maintains classified knowledge structures along with monitoring.

Chapter 4

Specification of the Resulting Model

Following the general design of AIM this chapter consolidates the resulting model by specifying its structures and functions. In accordance with the focus of the previous design, the specification is again devoted to those mechanisms composing the three parts—the representational framework, controlling mechanisms, and refining process—while others, though possibly important, are largely overlooked.

Moreover, the specification intends to make AIM a basis for various operational monitors so that further detail implementation could be conducted only with some technical concerns. The term *operational* means that a monitor yielded from AIM will perform with an acceptable efficiency regarding the ease of using and maintaining such tools. As a consequence, while concentrating on the main structures of those mechanisms, the specification will omit unnecessary details as long as the disregarded parts do not pertain to the main issues in this research.

4.1 Clarification of Functions

As designed in the previous chapter, AIM consists of five major components. This section clarifies functions to be provided for in each component so that each function's specification given later on can be traceable.

Representational Media constructs abstract representations of a monitored system. To accomplish this goal, it needs to contain the following functions:

- The function for generating and organizing constraint objects that characterize computing behaviors. Two types of objects—primitive objects and virtual objects—are involved in accordance with the previous design.

- The use of a virtual object in fact results from the configuration of a set of constraint objects. This requires the functions to configure several constraint objects together for composite monitoring. These functions need to deal with mapping, allocating, and binding monitoring objects.

- There are also needs for functions to further project a representation at an abstract layer to include emphasized semantic concerns along with variations on the descriptive resolution.

The functions of **Monitor Controller** are categorized into two parts: carrying the on-line filtering by application of constraints and executing the strategy of data collection. For these, the controlling part is generally composed of the following functions:

- Activating monitoring is the first function with which data collection may not always be operating during monitoring. Instead, data collection can be triggered based on the condition of a monitored system. This is necessary due to being conditionally operated, a circumstance which could reduce the running expenses.

- The function of default reasoning is the core ingredient for performing on-line filtering. The reasoning function is backed up by hierarchies of constraints and is guided with heuristics. The influence of heuristics on applying appropriate constraints in default reasoning is conducted through this function. An extension from this function is to guide monitoring to address certain parts by means of switching layers and varying details.

The functions of **Rule Refiner** focus on three types of knowledge, i.e., knowledge for filtering, for triggering collection, and for the refining process that improves the procedures of filtering and triggering.

- The first type of function is for maintaining a proper classification of constraints in terms of their maturity. The function evolves constraints with respect to the confidence weighted by heuristics.

- The next type of function is to refine the rules of controlling data collection. The rules for triggering and controlling—regarding the location, depth, width and frequency of monitoring—are adjusted and customized by this part. The function maintains the records from the feedback of monitoring.

- The third category of function refines the strategy of the refining process. This is accomplished by functions that adjust the weight of heuristics, the pace and direction of refining.

Although no detailed specification of **Acquisition Preprocessor** is specifically given, a moderately sophisticated interface is developed. So, some implementational details will be revealed in Chapter 6, where a prototype of this intelligent approach to monitoring is exhibited. The architecture of **Rule Base** is not explicitly specified, but the structure of constraints, constraint objects, and rules will be described.

4.2 Specifying Representational Media

This section specifies functions, which construct a fundamental framework for representing a monitored system, in the representational media. First, the knowledge entities involved in computation are generally abstracted into two sets:

Computing Units A computing unit is an arbitrary knowledge entity and could result from a mapping from a physical component, a logical component, or even a virtual component. A typical virtual component can be a policy which measures and manages computing activities. Thus a computing unit, mapped from a real computing component, possesses a certain degree of resolution and reflects certain semantic concerns. In addition, a computing unit may be viewed differently with different semantic concerns and with different levels of abstraction.

Configurations of Computing Units A configuration consists of a set of computing units and the relationships of these units. Logical relationships identify the concurrency, orders, and alternatives of interactions between two or more computing units. A set of computing units may yield a number of configurations in terms of different aspects.

Viewing various computing resources and activities in a unified manner eases the design of representational components. In the following, the representational framework is specified in a bottom-up approach, that is, from primitive components to macro structures.

4.2.1 Components of Representation

Fundamentally, a computer system can be described as a module consisting of a set of states, a set of relations, a set of initial states, and a set of events. That is, it can be thought of as an extended state machine. However, for reasons extensively dis-

cussed in Chapter 3, this representational framework is composed of constraint objects that characterize knowledge entities. Objects are constructed to feature computing activities generated in a monitored system such as performance status, liveliness, utilization, and others.

Events and States

For the purpose of abstracting computing activities, the traditional definitions of events and states are extended as emphasized below.

Events are considered internal and external as relative to the conceptual view of analysis. *Relative* means that an event may be understood as external in one observation and internal in another observation. The definition for external events is that an event is sensed by at least two logical components; whereas the definition for internal events is that an event is sensed by only one component. This concept is useful because it allows a physical event to be treated differently, depending on what level of abstraction an analyst intends to observe. Furthermore, events can be grouped to yield a macro event (also called a logical event), which may help characterize logics and functions at the level of application programs.

In addition, states are extended with respect to the preference chosen in each case of monitoring. Instead of being limited by physical states of a machine, a state can be abstracted by the preference of an analyst. Such a state can be obtained by projecting a physical state in terms of concerned attributes. As a consequence, a number of views in light of semantics may be perceived. In contrast to a macro event used to describe a complex concept, an abstract state is introduced to simplify physical states in a system. Nevertheless, the two treatments are designed as a result of the same motivation, which is to derive an abstract model from a monitored system.

Constraints

Constraints are primitive to objects, and objects are organized together to depict a monitored target. A constraint is composed of *terms*. Three types of terms are involved:

- a constant or a constant symbol
- a variable symbol
- a predicate expression, that is, if p is an n-ary predicate symbol, and $\tau_1, \tau_2, ..., \tau_n$ are terms, then $p(\tau_1, \tau_2, ..., \tau_n)$ is also a term.

In addition, the definition of a constraint is extended to consider temporal features. By being bound with T, which is a set of time symbols. The sets of variables and predicates are significantly dependent on time when coupled with T. Nevertheless, to achieve satisfactory performance in an on-line working environment, T is expressed as having abstract timing units [67], and dividing the whole time dimension into as few abstract timing slots as precision allows. For example, a constraint (possibly resulting from merging several constraints) may have such a form:

$$constraint = \begin{cases} c_1 & period_1 \\ c_2 & period_2 \end{cases}$$

In addition, conventional symbols are still reserved; when T is a singleton set, constants, variables, and predicates are disjointed from temporal featured behaviors.

Accordingly, a set of operations for combining terms to compose a more complex constraint is also expanded to include basic operations concerning temporal orders, such as $\preceq$, $=$, and $\neq$. The property of these three operations is stated as follows: if $term_1$ and $term_2$ are temporal terms, that is, their validity is limited by time, then $term_1 = term_2$, $term_1 \preceq term_2$, and $term_1 \neq term_2$ are temporal terms as well. The first formula is interpreted to mean that $term_1$ is valid as long as $term_2$ is valid; the

```
ObjectName:  coreMonitorObject {
    parentObject;
    /* link to an object at a higher level relating semantics */
    typeMonitor; /* atomic or composite */
    publicAttrbs {
        startTime;
        lifeTime;
        address;
    }
    proxyObj:
        <a set of proxy objects serving for this object>
    ctrlTbl:
        <a set of rules supervising monitoring>
}
```

Figure 4.1: The definition of the Monitor object class

second one means that $term_1$ and $term_2$ are valid at the same time; the last formula says that $term_1$ and $term_2$ are not jointly true.

Monitor Objects

Constraints are applied in an organizational manner. Constraints are grouped to yield objects where constraints exist as attributes and together characterize a target. There are fundamentally two classes of objects defined in AIM; one is atomic Monitor class and the other is composite Monitor class. The common part of these two object classes is shown in Figure 4.1. The specification for different parts of two object classes is examined in the next two subsections in which atomic monitoring objects and composite monitoring objects are specified, respectively. Each attribute in the class of Monitor objects is explained below:

StartTime is the time at which the monitor starts collecting data. If it is not specifically set, then the startTime of its parent is used. In the case of no parent object existing, the time at which the object is created is used as a default.

LifeTime is the period during which a monitor exists and is set by a default equaling the rest of its parent's life time if no preferred value is given. At the beginning of monitoring, the value of *lifeTime*, the period of time planned for observation, is set by an end user or by a system default and is adjustable during monitoring in terms of conditions. When a composite monitoring is timing out, all associated monitors terminate except those bound to other monitoring activities.

Address identifies an address at which collected data or the statistics of observation can be delivered.

ProxyObj is a linear table recording Monitor objects engaged in serving the Monitor object during operation. Among them could be atomic or composite objects. In the case of a Monitor object being atomic, this table points to probes that physically sense data for the Monitor object.

CtrlTbl gathers a set of rules supervising the essential behavior of a monitoring object such as ways of dealing with initialization at the start time, manipulating address, and cleaning up when its life time expires; they are not involved directly in data collection. The rules in the table are changeable; an example is allowing the most-frequently-used rules to remain in the table.

Atomic Monitor Objects

The class of atomic Monitor objects is defined as a child class of the Monitor class. In addition to inheriting most properties of the Monitor class, it has its own distinctive properties which are displayed in Figure 4.2. There are no private attributes defined in an atomic monitor since it does not contain any child objects so that there is no need to distinguish between private or public attributes.

ResourceType specifies a possible type of atomic monitoring. Among those predefined types are queues, stacks, variables, processes, buses, statements, routines, functions and others. Each primitive monitoring function is dedicated to one type of

```
ObjectClassName:  atmMonitorObject {
    coreMonitorObject;
    publicAttribs = {
        resourceType;
        semanticType;
    }
    publicMethods = {
        <methods for transmitting data, sending message, etc.>
    }
}
```

Figure 4.2: The definition of the atomic object class

monitoring for optimal efficiency.

SemanticType indicates interests about a target. Types of data required to be collected vary in terms of concerns which are expressed in the field.

Methods stores simple procedures directing the sending and moving of data, as well as communication with a controlling object.

No significant rules are used to guide an atomic monitoring object. The idea behind this is to make an atomic object as simple as possible. Also, it is designed to include temporal attributes in the atomic and composite object classes but not in the general class, (i.e., the Monitor class). This reduces the effort of maintaining the validity of each temporal attribute, since monitoring may not always be concerned with changes.

Atomic objects are defined to characterize primitive computing units such as variable, process, message, processor, bus, and others. To these basic types may be added more attributes to yield more specific computing units. The resulting structure is similar to the example shown in Figure 3.3. Some of these primitive computing units will be shown in Chapter 5.

```
ObjectClassName:  cmpMonitorObject {
    coreObject;
    <an execution body>;
    prvtAttribs = {
        resourceType;
        semanticType;
    }
    prvtMethods = {
        <a set of methods>
    }
    prvtControl = {
        ctrlTbl <a set of rules>
    }
}
```

Figure 4.3: The definition of the composite object class

Composite Monitor Objects

The above specification made for objects is addressed only at the primitive level, which provides small icons used to describe limited images but is underpowered in representing a complex system. To be capable of representing a monitored system, the representational model treats objects defined above as logical entities and further composes those entities. The class of composite Monitor objects is defined for this purpose. As a child class of the Monitor class, while sharing parents' attributes and methods, the composite monitoring class is designed to have more of its own attributes in a form as shown in Figure 4.3.

ResourceType specifies what task is to be monitored. This is similar to the counterpart defined in the atomic object class but with the difference that it may refer to a composite target.

SemanticType is similar to the counterpart in the atomic Monitor class but with one exception: it indicates more generally semantic concerns in comparison to those specific attributes that are identified in an atomic object. For example, whereas this field in a composite object may be used to express that the semantic concern of a

composite monitoring is the utilization of resources involved in a computation, the same field in an atomic object may be used to specify that the observation relates to the updating frequency.

A set of methods is an array of pointers indicating the methods available to a composite object. Among those, one method is mandatory for specifying the control over combined monitoring. The definitions of four types of combined monitoring are listed in the following:

- **Sequential Monitoring** specifies that the composite monitoring aims at a series of components resulting from the decomposition of a monitored system; each component is monitored and the corresponding data collection is processed in sequence.

- **Parallel Monitoring** means that the composite monitoring consists of a set of components that is a result of decomposition of a monitored system. Each component is monitored and the corresponding data collection is processed simultaneously.

- **Horizontal Monitoring** identifies monitoring as carried by a set of sibling Monitor objects, so that multiple-level monitoring is not involved. Monitoring at upper levels must halt while the monitoring—either in sequence or in parallel—is performed at an immediately lower level.

- **Vertical Monitoring** indicates that monitoring is carried by parent and child Monitor objects in parallel. This composite monitoring allows data collection to perform at more than one layer simultaneously.

Notice that vertical monitoring or horizontal monitoring is always paired either with sequential monitoring or parallel monitoring. Hence, composite monitoring falls into four categories (i.e., sequential horizontal, parallel horizontal, sequential vertical, and parallel vertical). When monitoring is conducted only at one level or when one object

is involved at a certain level, composite monitoring becomes a special case that is one of the above four individual types.

CtrlTbl identifies a controlling table with which a monitor often consults. It is updated by the monitoring controller. Due to the difference in structure and complexity between controlling tables in a composite object and in an atomic object, a controlling table is divided into two parts. One part is owned by its parent class and the other part is included in the composite Monitor class only.

An execution body is needed in a composite monitor because unlike an atomic object, a composite monitor indirectly collects data so that it must communicate with associated monitoring objects which collect data for it. A composite object follows rules to decompose a monitoring task into several subtasks and assigns such subtasks a number of atomic monitors and/or composite monitors. A composite object can be nested into another composite object when the need for further decomposing arises. The cooperation among a set of lower-level monitors is carried out through the algorithm encoded in the execution body.

Figures 4.4 and 4.5 together show the algorithm embedded in the execution body. It is designed to function based on a controlling table in a composite monitor, where the controlling table contains constraints. The intention of this design is to make the algorithm generic enough to fit into the controlling structure for most composite monitoring cases, and to result in less computing effort, as well as to enhance diverse applicability by heavily depending upon the constraints stored in the controlling table. The first part controls the creation of a new Monitor object—atomic or composite; the creating function is invoked—at the beginning and middle of monitoring—when a lower-level monitoring is needed. The second part controls the performance of monitoring. It also controls the cessation of monitoring at a certain level when no needs for monitoring exist any longer. Due to the fact that a composite object could be nested within another composite object, the above execution control may only be

```
If (a target does not match
    any predefined physical object)
then {
    consult with the ctrlTbl;
    if (a mapping to a set of subtargets is found)
    then {
        generate the matched set of child monitors;
        if (multi-level monitoring is not required)
        then
            stop monitoring at the current level;
    }
    else {
        if (the monitor is not the top one)
        then
            report to the parent monitor;
        else
            report to the monitoring controller;
    }
}
else {
    activate a matched physical monitor;
}
```

Figure 4.4: The part for generating composite monitoring

```
loop forever:  {
    listen for any calls;
    if (a call is received)
    case 1:  (from a child monitor)
        consult with ctrlTbl;
        take a proper action;

    case 2:  (from a parent)
        if (the message is recognizable)
        then {
            do whatever as told;
            send a message telling done;
        }
        else {
            send a message
            saying not understandable;
        }

    if (no child monitors are alive)
    then {
        report to the parent or the monitoring controller;
        terminate itself;
    }
}
```

Figure 4.5: The part for coordinating composite monitoring

part of the entire control. In this case, it communicates only to its parent object instead of to the monitoring controller. A dynamically generated composite object may not fit into predefined classes and is merely for simulating an abstract monitoring activity, so that it is called a virtual object.

The needs for allocating primitive monitoring objects and composite monitoring objects may be distinguished by the following example. The performance of a function may be observed by treating the function as a record-type variable with the difference being that its parameters are the fields of the record variable. The overall performance of an execution of the function can be roughly seen by observing this record. This type of observation is simply conducted by allocating an atomic monitoring object. As an alternative, it can also be monitored through observing the set of subfunctions which compose the function. So, a virtual object is created to bind those monitoring objects, each of which is dedicated to the observation of a subfunction.

4.2.2 Configuring Methods

Performing composite monitoring is accomplished via virtual objects. This section specifies the computational aspects composing virtual objects. The needed mechanisms are to deal with object equality and object composition.

Equality

The equality measured with the sign "==" over two objects is too restricting to be used for developing a flexible and semantic-dependent hierarchy which is important in simulating a highly dynamic system. From a conventional point of view, for example, "x == y" means that x and y must be the same type of variables as well as have the same value. To extend the strictly conventional equality, a mapping function is used to introduce a more abstract equivalence by considering circumstantial factors. A notation ∂ is used to indicate a contextual consideration. Accordingly, the above

equivalence expression is rewritten as

$$\partial\ (x \doteq y, semantics),$$

where *semantics* identifies the circumstance in which the equality holds. Through such a mapping function, x may be equal to y as long as they are the same type of variables, if the semantic field is indicated with "type," which means the equality at this time refers only to the variable type but not values or other factors. In another case, x and y can be considered equal if they are objects with the same inputs and outputs regardless of the possible difference in their internal structures, if the semantic field is marked with "interface." Based on this idea, five equalities are defined below:

1. Objects are equal if they belong to the same class. This definition says that two objects are equal in the sense that they have a certain number of common features and common methods even though they contain distinct, but not contradictory, characteristics. By this definition, two entities may be considered equal in a certain context if they are somewhat class-related.

2. Objects are equal if they have the same inputs and outputs when they are treated as black boxes. This equivalence is introduced to help mapping between different levels of abstraction.

3. Objects are functionally equal if they characterize the same computing activities in certain aspects. This is to say that when two objects are used to monitor the same targets with the same semantic concerns and even though one is not able to handle all monitoring tasks as another one does, these two objects are considered equal in terms of that part of their function. This equality allows a computing activity to be monitored by using those objects predefined for monitoring essential features.

4. One object is equal to a group of objects if one is functionally or input/output

equivalent to the group of objects. For example, when an object is formed by a concatenation of a number of other objects, the object is then equal to this group of objects.

5. A group of objects is equivalent to another group of objects if both can carry out the same task and can be viewed as equals by outside observers.

Paired with the equality, the notation defined to describe the inequality is shown below:

$$\partial\ (x \neq y, semantics),$$

Furthermore, the strength of equality is measured with the sum of each equivalence. Being equal in more aspects as listed above may make a greater degree of equality between two objects. Also, two objects can be equal and unequal at the same time but in different aspects.

Composition

Composite monitoring is developed through combining monitoring objects in terms of equality as specified above. Three types of operations—conjunction, unification and projection which are similar to relational operations [52]—are considered and are specified hereafter.

Conjunction Two conjuncting operations are defined and may serve for establishing relationships of parent-child and siblings between two objects.

Definition 4.1 Horiz-Concatenation Two objects, which characterize different tasks yet have one output incorporated into another, can be bound horizontally to have a sibling relationship. This is established by Equality 1, since these two objects are considered as having common attributes and at the same time having no contradicting attributes. Saying that there are no contradicting attributes means that part

of those attributes, which are different in each object, serve to describe unrelated features and thus result in no contradiction.

Definition 4.2 Vert-Concatenation One object can be vertically concatenated to another object or a set of objects with a *parent-child* relationship if they satisfy Equalities 2, 3 and 4. An obvious example is that an object describing a routine can be related by a *parent-child* relationship to a set of objects that characterize each statement of the routine.

Unification Unification is the type of operation making two objects cohere. Two operations for unification are specified.

Definition 4.3 Merge Two objects, which have partially overlapping attributes, that is, by Equality 1, can be unified into one object with additional processes on values of overlapped attributes. A special case is that if two objects describe completely different sets of attributes of a target, they can be unified into one object, which yields an addition between two objects.

Definition 4.4 Nest One object can be nested within another one if the nested object performs part of the functions that the nesting one can carry out. This is held by Equality 2. A useful application of this operation is to decompose part of a monitoring task and allow that part to be monitored in more detail.

Unary Operations Two unary operations are introduced. They are needed since, without them, it is difficult to describe situations such as selecting a subset of objects from a class, an object with fewer attributes, and others.

Definition 4.5 Projection An object can be derived by choosing part of the attributes from an original object with respect to a circumstance, preference, or other

reason. An object resulting from projection relates to the original object by Equalities 1 and 3. Division is not separately defined since it can be accommodated by projecting twice on two disjointed subsets of all the attributes contained in an object.

Definition 4.6 Selection This unary operation yields a subset of a class of objects by selecting those possessing special values in some of their attributes. Suppose that there is a set of constraint objects created to characterize arrival patterns. By *selection*, a monitor may choose some of these objects which feature specified patterns.

Two unary operations have some extended meanings in comparison to the concepts understood in relation theory. In this design, projection is to generalize a superclass while selection is to fork a subclass. An object generated through a unary operation is often relevant to an original object in a tight or loose manner, so that these two objects satisfy Equality 1 since they are class-related.

The above specified operations for composing objects can be grouped under another category: one to one, one to many, many to one, and many to many mapping. However, no operation is defined based on Equality 5 principally because these kinds of operations may not be practical, although theoretically they can be done. Moreover, it is generally possible (sometimes even preferable), to deal with a many-to-many relationship by processing a set of one-to-many relationships.

4.2.3 Constructing Abstract Layers

When accessed by a user from a preferred level, a representational model is defined through the four-parameter function shown below:

model(O, S, A, P), where

O is a set of objects at an immediate level,

S indicates semantic concerns,

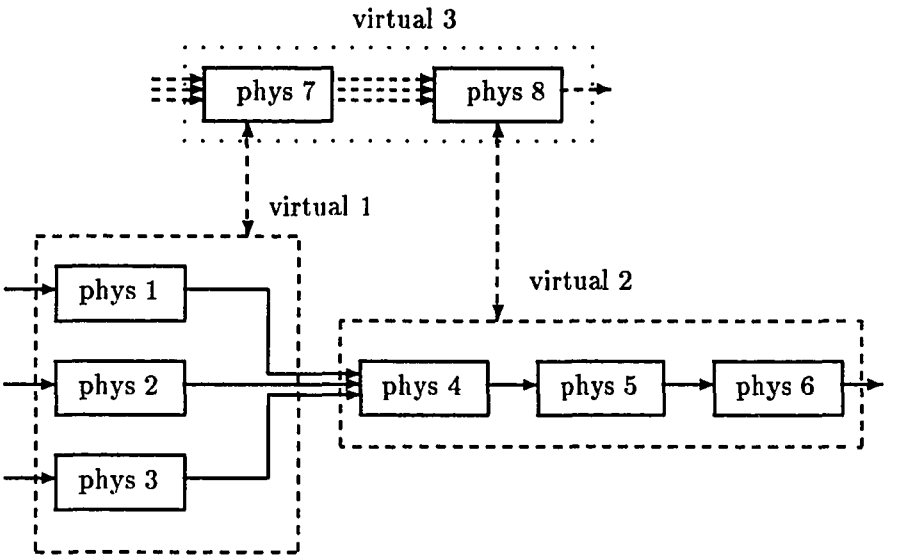


Figure 4.6: Converting a computing activity to a layered structure

A indicates levels of abstraction, and

P is a set of atomic monitoring objects other than those predefined.

Noting that these parameters are actually pointers to the specification made for a monitoring task, the model may dynamically be configured in terms of different monitoring cases.

A monitored system or activity is first isolated into abstract layers as discussed before. The construction of abstract layered representations is developed through computational media specified above. The specification for such decomposition is defined in general to follow these forms:

$compObj$ :: $(< compObj > | < atomObj >)OP(< compObj > | < atomObj >)^*$

OP:: CONJ | UNIF | PROJECT | SELECT

Consider a conversion of a monitored target to a layered representation as illustrated in Figure 4.6. From the top level, the virtual object (indicated in the figure as *virtual 3*, which relies on the abstract layer constituted by *phys 7* and *phys 8*), is defined through the following expression:

virtual 3:: (*phys 7 Horiz-Concatenation phys 8*)

The second-level decomposition is to detail the observation based on *phys 7* and *phys 8*. The mapping to the lower level is defined through *Vert-Concatenation*:

phys 7:: (*Vert-Concatenation virtual 2*), and

phys 8:: (*Vert-Concatenation virtual 3*)

Further, the detailed observation at the lower level is specified as follows:

virtual 2:: (*phys 1 Merge phys 2 Merge phys 3*), and

virtual 3:: (*phys 4 Horiz-Concatenation phys 5 Horiz-Concatenation phys 6*)

The specified connections between computing units imply that the constraints based on equalities may be enforced in monitoring. For example, since *phys 1, phys 2* and *phys 3* are merged, a sequential monitoring over these three computations may not be accepted.

All atomic objects to be nested must already exist; whereas all nested composite objects may already exist or may be dynamically generated. As long as a class to which each nested composite object belongs is predefined or can be found in a data dictionary, a composite object of this class is then created. Since definitions of an object class in a data dictionary are kept updated, the types of composite objects are redefinable.

4.2.4 Projecting Semantic Details

The preceding specification needs to be further extended to retrieve certain semantic aspects of a target which is viewed at an abstract layer. In terms of the discussion given in Section 3.4.3, the underlying orientation focuses on two types of collections, namely, data flow sampling and event-driven collection. Then the semantic concern can be further projected to filter unnecessary details at an abstract layer, a process which sets constraints and binds them to construct constraint objects for focused

targets.

Setting constraints can be a two-step process. The first selects the types of constraints and the second initializes constraints. While types are selected from the available ones, the value can be implicitly set by defaults. The defaults are expected to be found by matching a proper category of such defaults stored in a knowledge base. If such a match fails, the defaults' default is zero or a null character. The default's defaults will later on be set back by the result of the initial observation. Therefore, presuming the default values are set automatically, the following specification is only for selecting types of constraints.

In pursuing the logic-oriented monitoring, the selection of constraints follows the following procedure:

- The involved functions desired for observation at a chosen level of abstraction are selected, such as $[f_1, f_2, \ldots, f_n]$, where f_i is the name or address of a function. A function should be understood in a global sense so that it may be a statement within a module; the only difference is that the module has functional strength but the statement may not. Moreover, those selected functions do not necessarily cover all functions involved in performing the logic. The selection may only include critical functions that substantially influence others or are influenced by others.

- For each selected function, consider appropriate constraints that characterize attributes of the function to be observed. The time and output constraints are shown below:

$$f_i := (\text{time(constraint\_type, default)},$$
$$\text{output(constraint\_type, default)})$$

Constraint_type indicates which constraints are expected to be applied such as

long integer and float numbers. If the type of constraints is the type other than those predefined, an explicit definition is required.

In a resource-oriented monitoring, resources at an identified level of abstraction are selected and marked in the format: $[r_1, r_2, \ldots, r_n]$, where r_i is the identification or address of a resource. Secondly, constraints are selected to model the expected accessing patterns. In a situation in which a composite monitoring covers several resources, relative constraints are specified to constrain relations among those resources. In a similar sense of treating a statement as a function, variables are uniformly considered. In the category of variable-type resources are registers, stack, records, arrays, and buffers. Again, it is not necessary to include all resources which are used in the operation at an abstract level but merely those that are of concern. Finally, the resulting constraint objects for resource-oriented monitoring may have such forms as follows:

r_i:= (time_slot(constraint_type, default),
freq(constraint_type, default),
value(constraint_type, default))

And the same strategy of setting defaults for the logic-oriented monitoring can be used for the resource-emphasized monitoring.

Since monitoring essentially focuses on either functions or resources, the monitoring which concerns the performance of certain parts within an area is further situated in two basic types which are similar to the two orientations specified above; one considers those functions connected in a way that they directly invoke each other, and the other deals with those resources that are physically connected. One point worthy of mentioning is that the first type of geographical monitoring is different from the logic-oriented monitoring; functions are coupled but may not be necessarily related in logic. Typically, several functions, often existing as library routines and serving for

applicational computing activities asynchronously, may be dependent on each other and likely stay as a neighborhood. Although the selection of sets of functions and resources is still required in identifying the geograph-oriented monitoring, the difference is that the selection of functions or resources may focus on a chosen geographical area in contrast to the circumstances of the first two orientations of monitoring, which consider individual computing units, including algorithms and policies.

4.2.5 Setting Resolutions

After semantic details of a monitored target are decided and types of constraints are selected, the focus shifts to the degree of resolution in exhibiting those perspectives. As discussed in Section 3.4.3, the decision about the degree of resolution depends on criteria that are dynamically converted from a set of rules. A user is able to control the resolution upon individual cases. The controlling procedure, which periodically consults with the rules and decides on the degree of resolution, will be specified in Section 4.3.4. Hereafter, following the discussion given in Section 3.4.3, the underlying structure for changing resolution is detailed.

Fundamentally, the physical resolution is modeled by considering the basic types of data and linked in a manner shown in Figure 3.4.3. It is employed in case no specific structure is provided by the users. The format to identify a preferred resolution needs to include these attributes:

Type: (level_id, size, lower_level_id, higher_level_id)

where *type* indicates the type of data to be expected; *level_id* is the identification for the degree of resolution; *size* tells how much room is needed to hold data, and the default size is assigned according to *level_id* if *size* is not indicated. The predefined sizes may cover 1, 8, 16, 32, 64, 128, and $\lg n$ (n is the number of elements in a numerable set); the types of data may include *character(n), register, integer, float,*

record, and whether $n > 1$ means a string-type variable. *Higher* links the next higher resolution whereas *lower* links the next lower resolution.

On the basis of the above definitions, for each feature to be monitored, a number of different resolutions is selected and chained. Such a chained structure is fixed to basic types of data such as the time stamp (long integer), the visiting pattern (float), the percentage by 0..100 (sets or integer) or by 0..1 (subrange of float), and so on.

The structure of logical resolution is built up from the basis of the physical resolution. As discussed in Section 3.4.3, the logical resolution follows the levels dedicated to the collections of data, syntax, semantics, and up to models of monitored activities. As to the data collection, it collects primitive data without attention to synthesizing the meaning of collected data. It can be accomplished with the structure for varying physical resolution. When the syntax collection is demanded, more data structure is required for additional recording. The required specification for the higher-level logical resolution may include two indications:

set of data

type of relation

where *set of data* lists the data items required to be monitored as a group and *type of relation* specifies the association of data items in the set. The types of associated monitoring may cover *statement, message, function,* and *process.* This means that the data items identified in the set are associated because they are related by a statement, a module, a message packet, a function, or a process.

Monitoring at an even higher logical resolution may require the procedure of data collection to be specified. At this level, as discussed in Section 3.4.3, the meaning of data flows and the sequence of events are of concern. This is achieved by further binding information units in light of semantic relation and specifying constraints imposed on those units in a form like the following:

$$[inf_1, inf_2, \ldots, inf_i, \ldots, inf_k]$$

[relative constraints (if any)]

where inf_i is an information unit described with a lower resolution. Relative constraints may specify orders (e.g., sequential, parallel, or the degree of overlapping if neither strictly sequential nor fully parallel), input/output relations (e.g., the intersection sets), dependence on each other (sharing variables), etc.

Along with the ascending levels of observation, on-line filtering may need to allocate more statistical counters. The extra recording work is the result of the fact that the anomaly is not only judged by evaluating each observation but also determined by the pattern of a number of observed data, or even information, items. As a result, more counters are to be allocated for yielding mean, variance, distribution, and density of the observed performance. The strategy attempting to reach this type of resolution is considered below:

- In a normal situation, buffering those attributes that are influential in the correctness of later observation with a resolution as low as possible.
- In the case of no anomaly being discovered, the buffer will be re-used without dumping contents; when anomaly occurs, the buffer is flushed to off-line storage.

In monitoring a sequence of data or information items that are semantically bound, using buffers holds the preliminary observation to as low a cost as possible, and decisions of whether to store the observation are deferred until the whole sequence ends. Once some anomalies are identified, the resolution of buffering can be increased.

4.3 Specifying Controlling Mechanisms

The previously stated specification has presented an abstract model for describing a monitored target. One consequence of providing such a model is to make it possible to restrain data collection from becoming excessively detailed. Despite this, further

mechanisms for controlling data collection are still needed. To make the argument clear, two thoughts extracted from the previous discussion are stressed here:

- Most data may be considered insignificant, even though the data belongs to the appropriate types to be monitored. Significance means that data observed from a target can be used to distinguish the behavior of the target. Nevertheless, a large percentage of the observed data may not have much value in depicting the behavior of a target. For example, an event occurring with a normal interval may only contribute to a counter for this type of event, but the contents of such events may not always be worth storing completely. If such a decision can be made with sufficient confidence, the discriminating collection may significantly cut down the amount of data processed. The ratio of the room used to store an event identification to the room needed to store the data associated with the event is often very small. Hence, selective collection may further reduce substantially the amount of data to be collected while possibly still maintaining little disturbance to truth-preserving.

- Utilizing the proposed representational framework can possibly be achieved through on-line controlling mechanisms that may guide monitoring to switch to a proper abstract layer, to concentrate on distinct attributes, and to provide data with an appropriate resolution. Without these capabilities, the selection of all these aspects would only depend on end users and would not be changeable during the run time. This apparently fails to satisfy the demands from monitoring highly dynamic systems.

While the idea is exciting, the critical step is knowing how to collect sufficient information so that at a later stage, for example, the generation of a synthetic workload based on those data may mimic the nature of the encountered workload being monitored. The controlling mechanisms mainly provide two types of functions. The

first type of function measures the significance of monitored data through default reasoning and collects data accordingly, and is responsible for switching the layers, details, and resolution of monitoring. The second type of function serves for triggering control since monitoring should conditionally perform at certain parts and during certain periods. Since both types of functions basically rely on default reasoning, the structure and organization of default reasoning are defined first.

4.3.1 Structure of Default Reasoning

A default constraint is defined in the following, and a macro default that consists of a set of defaults is defined afterwards [30].

Definition 4.7 A default constraint, denoted as C_{dft}, has a form as

$$\frac{P : C_1, C_2, ..., C_n}{C}$$

where

$P, C_1, C_2, \ldots, C_n, and\ C$ are sentences or symbols;

P is called the prerequisite of C_{dft};

$C_1, C_2, ..., C_n$ are the justification of C_{dft};

C is the consequence of C_{dft} and is considered a constraint.

Notice that, in practice, sentences are often translated into special symbols so that the recognition can be quickly processed by simple comparison or computation. This design achieves the same end. In particular, Coften is a numerical symbol that stands for the special meaning of a constraint.

Defaults often need to be grouped to yield other defaults for reasoning a complex target. The definition of a macro default is given in the following.

Definition 4.8 A set of defaults, denoted as C_{dft} (δ) is then defined as

$$P(\delta) : C_1(\delta), C_2(\delta), \ldots, C_n(\delta)$$

$$\text{---------------------}$$

$$C(\delta)$$

where

$P(\delta), C_1(\delta), C_2(\delta), ..., C_n(\delta), and\ C$ are sets of sentences or symbols;

$P(\delta)$ is called the prerequisite of C_{dft} (δ);

$C_1(\delta), C_2(\delta), ..., and\ C_n(\delta)$ are the justification of $C_{dft}(\delta)$;

$C(\delta)$ is the consequence of $C_{dft}(\delta)$ and may be expressed with a set of constraints.

As discussed in Section 3.5.1, defaults are introduced as rules supplementing unsure knowledge and/or initial knowledge. Along with continuous monitoring, defaults are then refined and possibly enriched. Improvements on the reliability of defaults largely depend on associated rules that initially derive defaults. $C_1(\delta)$, $C_2(\delta)$, ..., and $C_n(\delta)$ are used in this design to bind those rules. In a normal situation, $C_1(\delta)$, $C_2(\delta)$, ..., and $C_n(\delta)$ gradually become more specific and/or more accurate so that $C(\delta)$ arrives with better confidence.

4.3.2 Basic Relative Defaults

One result of introducing virtual objects is the capability of monitoring the logical relations of associated computing activities. That is, constraint objects could be generated to apply relative constraints in observing the influence among a set of activities. The following are some relative constraints which are simple but useful in featuring the dependence of many types of concurrent computations:

1. $dft(x_i) = \min/\max(dft(x_1), dft(x_2), \ldots, dft(x_{i-1}))$, which says $dft(x_i)$ is computed based on the minimal or maximal default among previous applied defaults if no anomaly occurs by that point.

2. $dft(x_1) + dft(x_2) + \ldots + dft(x_n) == const$, which says that the sum of a group of defaults amounts to a constant. This type of constraint is used in characterizing related features of those activities that share limited resources, which may include the amount of computing time and the size taken in main memory.

3. $dft(x_1) - dft(x_2) = const$, which defines the two defaults which should retain a constant difference. It can often be used to prevent a monitor from collecting extra amount of data because of anomalies caused by related presequent activities. A simple case is one in which a delay is recorded but is actually caused by the preceding activities. Without use of relative constraints when such phenomena are seen, observed data may be misunderstood as anomalies caused by a current computation. With relative constraints, only the data relating to the activity that initially generates the phenomena is collected.

4. $dft(x_1)/dft(x_2) == const$, which fits into the situation in which two defaults may proportionally change. The time needed in accessing a file may increase proportionally to the number of files being accessed at the same time, if the overhead caused by switching from one job to another is negligible. In some situations, the logical relation between the service time and the number of files that are simultaneously accessed may break a linear relation. The point at which the relation becomes non-linear is then considered a triggering point for data collection.

5. $dft(x_1) * dft(x_2) == const$. The most important use of this constraint could be the application of *Little's theorem*. For example, in the case where a number of customers is constant, the average amount of time spent by customers in the system may be inferred from the default of the arrival rate through this relation.

Relative constraints along with the use of virtual objects strengthen the capability of collecting critical data in a concurrent system. Some typical applications relevant to

the above specified relative constraints are examined below. In observing a mutually exclusive computation, two related computing units may be bound by a relative constraint with the form $dft(x_1) + dft(x_2) == const.$ If a computation requests access to a forbidden area, *const* can be set to 1 and $dft(x_1)$ is either 1 or 0 toggling with $dft(x_2)$. These two monitoring objects are bound by a virtual object where a 1-bit room is allocated for both defaults. As an entry signal occurs (assuming 1 is issued upon an entering or exiting signal), the value in the 1-bit room flips. If the result is 0 after one enters, then it tells when a violation to the accessing rule happens. Consequently, more details may need to be recorded.

In the case of monitoring a producer/consumer computation, the default for the computing unit on behalf of consuming is set to the buffer size subtracted by the default applied to observe the producing side (i.e., *const* - $dft(prod)$ and *const* is the size of a buffer), and vice versa. The case can be extended to consider a group of producers and a group of consumers. In such a situation, in addition to applying this relative constraint, within each group, another relative constraint with the form $dft(x_1) + dft(x_2) + \ldots + dft(x_k) = func(dft(prod))$ is required, where $func(dft(prod))$ equals *const* - ($dft(prod)$ and $dft(prod)$ stands for a constraint on the group of producers.

Without the use of relative constraint and the support of virtual objects, monitoring would not be able to detect this type of anomaly unless comprehensive data collection and subtle off-line analysis were applied. However, as noticed in the preliminary study, the complexity of modern computer systems often makes comprehensive monitoring prohibitive.

4.3.3 Order of Defaults

Defaults are discriminated in accordance with assigned priority. The order of attempting a group of defaults that are to characterize the same feature is determined through heuristics. A possible method of applying heuristics to determine the priority

of each default among a group of defaults is specified below:

- Choose a proper subset of heuristics from the set of available heuristics:

  $[h_1, h_2, \ldots, h_k]$.

  Referring to the list of heuristics in Section 3.5.1, the above set may stand for those heuristics, namely: *most often used*, *least computational*, *least covered*, *most anomalous*, and *oldest*. These symbols may be just identifications, each of which points to the place where the definitions of these heuristics is stored.

- Assign a weight to each heuristic so that the total weight distributed to all applicable heuristics amounts to *1*. These different weights indicate the discrimination among these heuristics and are accordingly assigned to each heuristic. The assignment may reflect the preference of human experts.

  $[(a_1, h_1), (a_2, h_2), \ldots, (a_n, h_k)]$,

  where $(a_1 + a_2 + \ldots + a_n) == 1$.

- Attach a coefficient to each weight (i.e., a_i). This type of coefficients is used to suppress the strength of a corresponding heuristic. Each coefficient is initialized with the same value that may be an average between the upper bound and the lower bound (e.g., 0.5) if no explicit value is given by a user. The resulting form is shown as follows:

  $[(b_1 * a_1, h_1), (b_2 * a_2, h_2), \ldots, (b_n * a_n, h_k)]$,

  where $(b_1 + b_2 + \ldots + b_n) == const.$

  Const means that later on in monitoring, b_i may change but their sum will be constant.

As monitoring is operating, more constraints may be generated and b_i will be adjusted. Possibly, some newly developed constraints may have greater weight than the old ones because b_i may be increased to a degree overwhelming the influence of a_i set initially, thus the new ones will be applied instead of the old ones.

The weighted defaults are organized into a partial-ordered structure. This is because, for example, a default may compare with two other alternate defaults while these two defaults may not be comparable for various reasons; for instance, they may never be jointly true.

A simple case is a default constraint featuring the service time during weekdays with two alternate defaults, one for Monday and the other for Friday. The two defaults for the special estimation over the Monday and Friday performance have no relation to each other, so that the priority between these two is not comparable.

4.3.4 Controlling Procedure

The controlling procedure is composed of two parts: one is for applying constraints and the other is for triggering the filtering.

Controlling for Collection

Figure 4.7 shows an algorithm that controls data collection through default reasoning. The default assumption makes the algorithm capable of isolating complexity at each reasoning step so that the method can be operational with respect to limited computing time. Anomalies mentioned in the algorithm are cited in the following but the list should not be considered inclusive:

- phenomena that have not been seen for an abnormally long period,
- values that touch or are near boundaries for certain data flows,
- phenomena that occur with unknown patterns,

```
LOOP {
  observing outputs from active constraints objects
  switch ( anomalies occur at a certain degree ) {
  if case 0:  ( alternative constraints exist )
     then ( select proper ones based on the order )
  if case 1:  ( physical resolution can be adjusted higher )
     then ( increase the physical resolution )
  if case 2:  ( logical resolution can be adjusted higher )
     then ( increase the logical resolution )
  if case 3:  ( semantic details are not fully observed )
     then ( allocate additional constraints for observing them
  if case 4:  ( the lower-level observation is specified )
     then ( allocate child objects for the lower-level observation )
  }
  register the level, semantic details, resolution;
}
```

Figure 4.7: The algorithm for controlling data collection

- types of data that appear irregularly,
- changing frequencies in an abrupt manner.

Controlling on Triggering

Data collection should not be constantly operating. Thus, in addition to employing algorithms for filtering observed data, part of the controlling functions also controls what time and under what conditions the data collection should be kept operating. Rules supporting these decisions are composed of both domain-dependent and domain-independent knowledge. Typically, within an adequately long period of time, if a monitored target is observed to be normal, monitoring may autonomously adjust as follows:

- to maintain observation at more abstract levels,
- to slow down its sampling frequency,

- to suspend data collection periodically, or
- to collect data only when triggered upon certain conditions.

The suspension of monitoring may be issued to certain parts of an entire target if the parts satisfy conditions identified above. That is, monitoring may conditionally be suspended locally.

The conditions for suspending data collection are defined by certain rules so that such conditions can possibly be changed in individual cases. At the beginning, rules are largely composed of domain-independent knowledge and are often heavily statistics-oriented. Then the adjustment of those rules proceeds in terms of the performance of monitoring, which is to get accustomed to a system being monitored. The following are some of those rules that may control the switching of data collection:

- if $x\%$ of the observed data is within a mean value $y \pm \epsilon$, then pause for z seconds;
- if interarrivals follow a pattern as predicted for an x long period, then pause for y seconds;
- if a newly-forked lower-level observation covers the observation at higher levels, then the observation at higher levels may be halted;
- if certain sections have clear boundaries and satisfy one of the above rules, then those sections may be less detailed or less frequent until suspended;
- if a section, A, is independent of other sections—$B_1, B_2, \ldots, B_n$—in a sequential monitoring, that is, $P(A) \doteq P(A|B_1) \doteq P(A|B_2) \doteq \ldots \doteq P(A|B_n)$, then Section A can be monitored separately or monitoring on Section A halts while Sections $B_1, B_2, \ldots, B_n$ are monitored;
- if an arrival pattern follows a uniformly random distribution, monitoring may halt at any point as long as one of the above conditions is matched.

Rules for triggering data collection are certainly more than the ones just instanced, but these rules are expected to be frequently used. Notice that these rules are often parameterized; this again permits suitable values to be filled in when these rules are applied from case to case.

4.4 Specifying Refining Function

Defaults and structures of defaults must be changeable by means of methods discussed in previous chapters. The mechanisms integrated in the rule refiner for conducting the evolution is therefore specified hereafter.

4.4.1 Classification of Knowledge Entities

The basis for evolving the knowledge invoked in monitoring is the classification of defaults and constraints. The classification applied in this context is semantically different from a representational hierarchy described in Section 4.2. The former is to classify constraints in terms of semantic meaning and a set of criteria and is maintained by the monitoring refiner for the improvement of data collection; the latter is for representing a monitored system, which is organized by the monitoring controller and relies on the understanding of system structures. The knowledge entities are essentially distributed among the following classifications:

- Exceptional Primitive Concepts. This class corresponds to newly created and non-decomposable concepts. Whether a concept is decomposable is relevant to the level at which a problem is viewed.

- Exceptional Combined Concepts. In contrast to the first class, this class refers to those concepts that are composed of other concepts. However, at least some of the knowledge entities in an exceptional combined concept are considered

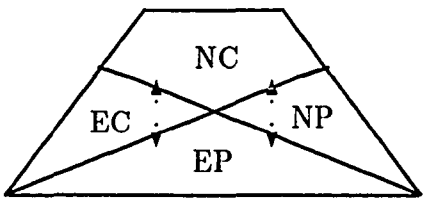
exceptional.

- Normal Primitive Concepts. This class is to identify those concepts that have become matured and are not decomposable in terms of levels of abstraction to which they refer. A concept in this class is composed of a fixed number of attributes for characterizing certain computing units or their relations. The distinct property of this type of entities is that those concepts are unlikely to be doubtful and are not allowed to be eliminated. If some doubt is finally discovered, the consequence is that the referred entities are dropped to the class of exceptional primitive concepts.

- Normal Combined Concepts. Accordingly, this class is used to specify those stable concepts which consist of several other concepts, either primitive or combined. Similarly, the knowledge entities in this class are much more trustworthy than those in the class of Exceptional Combined Concepts.

4.4.2 Refining Paths

There are totally ten paths designed for promoting and demoting concepts distributed in the above depicted classes. Figure 4.8 shows refining paths over the classified concepts. These ten routes conduct the evolution by promoting knowledge entities from null to Exceptional Primitive Concepts; from Exceptional Primitive Concepts to Normal Primitive Concepts or to Exceptional Combined Concepts; from Normal Primitive Concepts or Exceptional Combined Concepts to Normal Combined Concepts; and by demoting knowledge entities through the other five routes in the reverse direction. All entities are organized into a tree-like data structure; each entity is possibly moved from one position to another inside the tree structure via the refining routes. The definition of these paths, including the explanation about operations conveying each movement, are given below:



NC: Normal Combined Concepts,
EC: Exceptional Combined Concepts,
NP: Normal Primitive Concepts,
EP: Exceptional Primitive Concepts,
: Refining Paths.

Figure 4.8: Knowledge classification and refining paths

- Creating an exceptional concept. That is, the path leads to raising the creation of a concept. For realizing such a creation, the refining process records the observation by creating a record tagged with Exceptional Primitive Concept, and also identifies the relationship between the newly created structure and existing structures. With a hierarchical structure, the newly created concept is usually inserted at the bottom of the hierarchy which stores knowledge entities that describe similar concepts.

- Deleting an exceptional concept. This path takes a reverse direction compared to the above path; it results in a deletion of a leaf from the hierarchy. Note that this implies that only extremely exceptional cases are possibly dropped off.

- Promoting an exceptional concept to a normal concept. This requires the refining process to identify the concept as a normal concept and to adjust the position of the concept in a hierarchy. If this promotion leads to a conflict in another normal concept, then further adjustment between these two concepts is required. This process produces another, the definition of which follows.

- Dropping a normal case to an exceptional case. This normally requires a refining process to find another substitution. In a hierarchical structure, the alternative entity may usually be picked up from the location at an immediately lower level.

- Promoting a primitive concept to a combined concept. A concept is simple at the time it appears. Along with monitoring, related concepts may be deliberately grouped as a combined concept. At the creation of a combined concept, a node is generated and inserted into a hierarchy wherein related combined concepts are properly ranked. The structure storing a combined concept includes a head and pointers referring to participating concepts.

- Breaking a combined concept into one or more simpler concepts. The path is a two-step process: the correspondent structure is removed from a hierarchy organizing combined concepts; then each component is examined to see whether further demotions are needed. If they are, further demoting actions are triggered and lead to another two-step process. This process may recursively be carried on until either each component remains in its original class or a primitive level is reached.

Types of Evolution between Exceptional Primitive Concept and Exceptional Combined Concept as well as between Exceptional Combined Concept and Normal Combined Concept are conveyed through another four paths that are similar to the paths described above. These mapping paths and the early-on specified classes together hold a closure for knowledge evolution.

4.4.3 Refining Criteria

In addition, a set of generic heuristics is provided to serve as the refining conditions. Strictly speaking, a refining procedure starts at the time when a statistical accumulation begins to record significant data that have been observed, or such statistical

recording may periodically be turned on. As mentioned before, not all recording work may finally trigger a refining action. As a matter of fact, most accumulated recording work is eliminated before it is backed up to the degree at which a refining action may fire. The degree is determined by the refining criteria. Criteria used for classifying knowledge entities are expressed through declarative knowledge, and therefore they are adjustable in accordance with individual systems and cases. Refining criteria are divided into two sets listed as follows, one for promotion and the other for demotion:

- The set of criteria for strengthening knowledge entities could be these:

    1. If a newly created knowledge entity or an existing knowledge entity that contradicts a more mature case does not worsen the performance of a monitored system, then the entity may be accepted as an exceptional case. In other words, the case must appear stable and its existence must not damage the performance of other parts.
    2. If an exceptional case is found more useful and more convincing to a certain degree in describing the same concept than a corresponding normal case, the exceptional case is promoted to normal status whereas the normal case becomes exceptional.
    3. If one exception happens more often than another during a certain lengthy period, then the two exceptions switch the priorities being applied.

- The criteria for demoting knowledge entities are enumerated below:

    1. If an exception has not been met for quite a while, then it is deleted. Such deletion is allowed only at the bottom of a case tree in order to ensure that a deleted case is the one least frequently confronted in comparison to other similar cases, and it has not been used for a period of time.
    2. If an exception is upgraded, then there must be at least one exceptional case or a corresponding degraded normal case. This criterion is used in

part to insure only one knowledge entity being considered normal among a set of entities describing the same target.

3. If an exceptional phenomenon has made the performance of a monitored system unstable, it is demoted by switching its position with the one ranking at the next lower level.

One implication from the above defined criteria is that refining criteria are basically composed of domain-independent knowledge. It is in fact very much statistically oriented. In spite of this, some parameters used in criteria are dependent on each individual application. For example, how long a period should be in order to yield a reliable outcome from observation is heavily dependent on patterns of the working load of a monitored system.

4.4.4 Refining Process

The refining process is dedicated to three types of refining—refining constraints for characterizing monitored systems and refining rules for collecting conditions and changing the strategy of refining. The first refining process is in fact responsible for physically changing default hierarchies by measuring observed phenomena in terms of refining criteria. This refining process ignores what the semantic meanings of those nodes in a hierarchy are and makes judgements based on comparisons among performances of those nodes. The refining process is situated on the basis of the above specified evolutionary structures.

Since constraints represent defaults that are organized into the tree-type structures, each type of refining process is designed to maintain a fine organization of defaults so that default reasoning may be applied effectively. The first type of refining focuses on each hierarchical default structure that characterizes a concept, or, say, a monitored activity. The constraint object at the top of the hierarchy is con-

```
Passing observed and collected data
switch ( which_case( data ) ) {
    a known case:  updating a correspondent statistical record;
    default:  recording it into a new record
              with a unique identification;
}

for (each evolving path) {
     comparing the record with the condition set for the path
    if ( comparing result is positive ) {
          promoting or demoting corresponding constraints
          or constraint objects as indicated;
    }
}
```

Figure 4.9: A refining algorithm for updating a default hierarchy

sidered default and is most often applied. Underneath it, there are all the possible substitutions of the default. These substitutes are positioned in accordance with their assigned priority. Such a default structure is maintained by three methods. One promotes and demotes each entity in a default hierarchy; the next improves a knowledge entity in the hierarchy and does not necessarily change the position of the entity; the last is to adjust criteria which are used to measure effectiveness of each knowledge entity. The following specification clarifies these three methods in detail. The algorithm to carry out this refining process is given in Figure 4.9. Before taking a look at the algorithm, it must be pointed out that constructing a default hierarchy suitable for on-line filtering could be a complex task due to the proper selection of referenced heuristics. Thus, computing efforts involved in figuring out the strength of each heuristic also vary to a large degree. In coping with this difficulty, the refining algorithm is designed to be more strategic instead of technical in detail. This allows the algorithm to include important aspects that ought to be of concern while ensuring that the further development may not be incompatible with this basic structure.

The refining algorithm is constructed to fundamentally follow three steps: detecting, complying, and confirming. The refining process checks on any significant phenomena passed from the collecting process. The significance is determined by the principles described in Section 4.4.3. Once conditions embedded in one of the evolving paths described in Section 4.4.2 are satisfied, an evolving action starts by following the corresponding path. At the beginning of the existence of an object, it stays only in a group called *exception.* It may be finally moved into the *normal* class if the case is proved to be normal.

The confirmation is naturally performed by two processes:

- Adding, dropping, or modifying a case does not happen when a violation is first detected. Instead, at the beginning, an anomaly is only recorded and the occurrences of anomaly accumulate. Then, the consequence could be two possibilities: either the accumulation finally triggers a refining process to accept the related case as an exception, or the accumulation is conditionally stopped if similar phenomena are no longer detected. An object is allocated at the bottom level when the object is created. Hence, the procedure of moving from the bottom level to the level near the top actually conducts a confirming process.

- Crossing the boundary of normal and exceptional classes requires substantial evidence to show the stability of a case. This means that entering into the normal class may require more observation.

The refining algorithm basically functions upon the guidance of a collection of heuristics and refining criteria that are essentially declarative knowledge. An instanced structure resulting from the refining algorithm is illustrated in Figure 4.10.

The second category of refining is to tune controlling rules. The main functions accomplishing this type of refining are specified as follows:

- Frequency of data collection is to be adjusted, not only according to individual

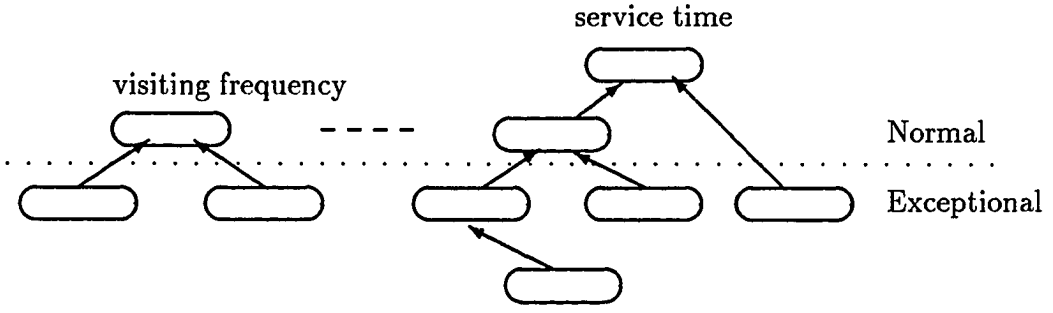


Figure 4.10: The resulting structures of classified constraints

systems but also varying from time to time when monitoring the same system. Nevertheless, strictly speaking, an optimal sampling frequency would only be made possible by support from effective pattern recognition. And the pattern recognition should be considered as the task of off-line analysis due to the fact that such a learning process involves heavy computation and may not always be convergent. Yet the results derived from the process of pattern recognition of work loads should be utilized to benefit the decision-making of monitoring frequency. To leave such an opportunity open, the triggering control relies on a set of rules. Collecting conditions can be customized to individual cases by updating rules to fit into changing situations.

- The criteria for considering anomalies are designed to be changeable. Although many of those rules as stated previously are generally applicable, the effectiveness of each rule for each individual case still may vary dramatically. The adjustment of those criteria in principle follows the method explained below. A criterion may be yielded through an equation such as:

  $\text{conc} = \alpha_1\beta_1 + \alpha_2\beta_2 + \ldots + \alpha_n\beta_n$, where

  α_i is a weight parameter and β_i is a function on behalf of a rule or a set of rules.

The equation uses an additive variation to combine related rules. To discriminate the strength of each rule, the refining method is responsible for adjusting values of each parameter so that the influence from the associating function varies accordingly. More specifically, assume that β_1 is a discrete function specifying mean service time of each weekday; β_2 is also a discrete function giving a predicted mean based on hours within a day; β_3 then describes the mean in terms of sections in a semester, similar to the rest. The influence of β_i derived from a correspondent function depends on α_i. Due to the possibility that the work load of a system may be changed because some courses are redesigned to distribute their load differently—e.g., cancelling some term projects and adding more daily homework—weights have to be updated to match patterns of changed distribution in course work; some of the factors become more influential while the weight of others' influence is reduced.

The third category of refining is to change heuristic rules which decide the priority of constraints. These principles are the knowledge ultimately affecting the parts of constraints to be applied and how they should be applied. A distinct feature of refining heuristic principles is its slow motion. Although it is hard to make this part of the knowledge purely declarative, it is desirable to maximize the declarative strength as much as possible. The refining of this aspect of knowledge includes the following processes:

- Adjusting the weight of each heuristic so that a heuristic has an adaptable influence in making decisions about which should be monitored and to what degree. The emphasis on which heuristic heavily depends is the orientation of monitoring. Thus the changing of weight, which a heuristic pertains to, has largely to follow individual needs. Taking a look at the *least covered* principle: It should be preferred if one wants to quickly find out fault resources. However, in the case of diagnosing abnormal components, the *least covered* measure may

not be appropriate for use in the beginning of such a diagnosis; the *least covered* principle usually leads to isolating the part where originating faults occur but not to the area being influenced. Hence, the influential strength of each heuristic should be adjusted in light of context. In another situation, the weight of a heuristic could be zero. That is, the heuristic would no longer be influential. An opposite situation, though unlikely, is to allow one heuristic to have a decisive weight in deciding when and how to apply constraints during monitoring.

- Adjusting the pace of increasing and decreasing the weight of each heuristic to prevent part of evolving knowledge from having extraordinary strength. For these heuristics that already have a high strength of influence on the triggering of certain sets of constraints, some resistance ought to be gradually inserted so that other heuristics are not ignored. This is needed particularly because of two reasons: heuristics can be quickly customized from one special monitoring environment to another, and this adjustment may avoid hiding fallacies in an unbalanced system.

- More meta-flavored refining is to adjust semantic meaning of a heuristic, namely the definition of a heuristic. However, this design leaves this type of change to be incorporated through the intervention of human experts.

4.5 Clarification

Up to now, the specification of the proposed model has been given. Mechanisms fundamentally constituting the resulting model have been specified in such detail that they are capable of depicting a sufficiently clear structure so that further implementation of a monitor will only be concerned with technical details.

A few important aspects of developing a monitor are not exhaustively discussed because they either fall outside of the scope of this research or are technologically

resolved by this time. Among them are primarily the following:

- The design of monitoring probes is of no concern. All monitors need sensors to detect signals no matter what approaches they are adopting. The technology for implementing those probes has reached a sophisticated level. Since there is no significant distinction between sensors used in this and other approaches, this part of the design is left out of this specification.

- Some computing activities that recursively nest with each other do not receive special consideration. In consequence, such activities may be treated as nondecomposable computations; thus, the model may be limited due to its capability of only representing such computations as a one-layer abstract model.

- The synchronization of time and clocks referred to by different computing units in a computer system is assumed to be tolerable for setting time boundaries of temporal constraints.

- Some parts of the resulting model do not have many notable features with regard to the contribution in the monitoring area, so the details of these parts have not received much effort. However, necessary components within those overlooked parts will still be implemented in the prototype of the resulting monitoring model and will be examined later in this thesis where the prototype is presented.

Chapter 5

Applications

The preceding chapters have presented AIM, a model for intelligent monitoring. An ideal way of showing the feasibility of the resulting model and making the resulting advantages convincing is to go through concrete examples which apply AIM to actual monitoring cases. To achieve this purpose, some considerations ought carefully to be taken into account. First, the demonstration should show that AIM performs with the claimed superiority in monitoring computing activities with such attributes as:

- their behaviors may be highly dynamic;
- their environments are too complicated to be easily understood by users;
- their complexity makes comprehensive monitoring prohibitively difficult.

In addition, the usefulness of the researched monitoring model also requires demonstration of such evidence as:

- AIM does not shift the complexity to on-line users. Instead, achieved improvements largely depend on the support of intelligent capabilities, in particular the reliance on declarative knowledge.

- AIM is applicable to monitoring diverse computing activities and is easily customized for a specific monitoring task; the manipulation of such a model is moderately difficult.

Furthermore, a computing activity chosen for illustration must satisfy two other conditions: one, that such an activity should be often confronted in using computer systems; and the other, that the application of AIM to such an activity is relatively inclusive.

With these considerations in mind, the case of monitoring a file access in a UNIX-like environment[1] is selected as an example. For exhibiting the primary research results, the demonstration concentrates on the capabilities of AIM in the following aspects:

- providing significant data,
- enabling monitoring with regard to various concerns,
- supporting effective assistance in fault-detecting, and
- behaving adaptably in accordance with the change of a monitored system.

The evaluation of the resulting monitoring model through this concrete example again concentrates on three major aspects considered in designing AIM, i.e., representing the monitored target, controlling data collection, and refining knowledge of monitoring.

5.1 Case Description

A brief description of the monitored activity is given in this section, which includes primitive concepts and assumptions about the file access.

[1]UNIX is a trademark of AT&T.

5.1.1 Case Related Concepts

A file is defined as a collection of data elements grouped together for purposes of access control, retrieval, and modification. A file is perceived to contain an ordered collection of records. For each file, there are sections storing descriptive information specifying the properties of the file. A file access is an activity supported by a file system and typically can be one of these actions: creating, destroying, organizing, reading, and writing.

A UNIX-like file system supports a group of users who may access files asynchronously. Since keeping track of all files in a single directory is impossible, a UNIX-like file system uses a flexible hierarchical directory structure [25]. As a result of such a tree-like structure, identifying a file requires a complete path name beginning with the root directory or a user's current directory. A set of access procedures is provided to support the above mentioned access types.

In a UNIX-like system, a request to access a file is issued from a system shell or from an application program. The file system inspects the command against those permitted types of actions. If this is passed, then a temporary file control block is created and is used to keep track of any changes made on the file. After the access is completed, the file control block is closed and any permitted changes can now be realized.

5.1.2 Assumptions

To focus attention on major issues addressed in this research, some assumptions are made to simplify the case in some aspects while stressing other important aspects. These assumptions are not essential to the success of this study but are merely introduced to avoid distraction by unimportant details. There are essentially three assumptions:

1. The examined system is supported by a collection of processors managed by a central control.

2. Processes may execute asynchronously without restriction whether or not multiple processors are provided.

3. Clocks have negligible time delay between each other within the monitored system.

5.2 Representing the Target

As specified in AIM, a monitoring task is composed of three major processes, namely representing targets, controlling data collection, and refining the behavior of monitoring. This section describes the first process, which is to represent activities involved in a file access. Following the procedure designed in AIM, representing the target consists of two steps. First, primitive components needed for constructing the representation are developed; this can be done either by selecting from predefined components or by creating them. Second, the make-up of components is developed to simulate the target in terms of logical or physical relationship. These two processing procedures are examined in the next two subsections, respectively.

5.2.1 Representing Basic Computing Units

In general, monitoring a file access can be described fundamentally by a set of primitive computing units such as processes, functions, statements (which can be a loop statement, a conditional statement, or an assignment statement), and basic data structures including a queue, a stack, an array, and a single variable. General characteristics of these components should be common across machines and applications. With these primitive components being defined as classes, a set of constraint ob-

jects can be generated directly by the monitor. In the beginning, these constraint objects are expected to be very general regarding their characterizing ability, since the knowledge applied in generating these constraint objects may be from qualitative analysis, namely, the understanding of similar systems and theoretical computation. Further, the knowledge may gradually become more specific by being consolidated with quantitative data gained through monitoring and analyzing. As for a file access, the needed components with their common attributes can be highlighted as follows:

- **Stack** Feature: first in and last out and size can be changed;

  Access: pop up and push down;

  Typical errors: overflow or down flow.

- **Queue**

  Feature: first in and first out, changeable size, and it can be doubly-chained or be extended to a ring;

  Access: add and delete;

  Typical errors: the chain is broken.

- **Array**

  Feature: convenient to access and fixed size;

  Access: random and sequential accesses;

  Typical errors: an index to an array is out of the boundary.

- **Variable**

  Feature: effective scope such as local versus global;

  Access: copied by address or by contents;

  Typical errors: type mismatch and improper initialization.

- **Function**

  Feature: having input and output; expressed through a routine;

Access: invoked upon a call;

Typical errors: mismatched inputs and unexpected outputs.

- **Process**

  Feature: having a lifetime and a parent; existing in one of states such as active, waiting, and suspended;

  Access: forked out by another process; terminated when either its lifetime expires, or it is killed by a parent process or the operating system;

  Typical errors: abruptly terminated, or staying in one of the states for an unusually long time.

As designed in AIM, the above primitive components can be organized in relation to each other. Typically, the *variable* component can be considered more general, since its features are ordinarily common to other components. Hence, primitive components are possibly connected in such a way that some may inherit properties of others. Because the above instanced components are fairly primitive, it is reasonable to expect that the classes of these components are predefined in a knowledge base and are ready for retrieval by an individual monitoring case.

With these primitive components defined, the next step is to organize them to represent the monitored activity, namely a file access. As defined in AIM, deciding the level of abstract details at which the monitoring starts should be the first concern. Once an abstract layer is set, composing abstract constraint objects is then accordingly accomplished. This includes consideration of the granularity of observation at this layer, selection of functions from those observable at this layer, designation of attributes to be looked at for each selected function, and identification of the range of degrees of resolution for describing each attribute. The whole procedure is intermingled. Hence, in real cases, layers, details, and resolution are dealt with without a clear timing order. Accordingly, the following exhibition of representing a file access is based on the development of constraint objects instead of discussing layers, details

and resolution separately as was done in the preceding chapters.

5.2.2 Representing Abstract Activities

In using primitive components, one can directly apply them or combine several of them together to characterize computing units involved in a file access. This process is accomplished on the basis of viewing a file access abstractly at various levels.

In the abstract sense, as seen from the point of view of an end user, a file access is typically composed of a set of generic operations supported by the system, that is, a virtual machine extended by an operating system. A user usually does not see, or perhaps often does not need to care about, those details that are underneath macro commands. Therefore, the monitoring may initially focus on the level of such an abstract user interface. The typical set of operations at this level consists of creating, opening, closing, reading, writing, and deleting [6, 13]. Each of these operations must be converted to the underlying procedures defined in the file system, and each is finally transformed into the device-dependent commands; by treating each operation as a non-decomposable computing activity, the monitoring may not be able to sense all details but only the signals that indicate the success or failure of these operations.

In terms of AIM, there are several ways to represent the performance at the level of user interface abstraction. Figure 5.1 and Figure 5.2 depict two alternatives. Figure 5.1 illustrates an abstract representation at the top level. Monitoring with this model may only collect a small amount of data such as a given file name, an access request, and a returned number indicating the amount of data accessed. Also, the time spent for file access may be recorded, but no more details will be given. The constraints which are selected among those that may characterize the same feature are determined by the desired degree of resolution and semantic detail. In terms of selected resolution and details, some data would either not be collected or might simply not be seen. To clarify this point, assume that an analyst wants to see whether

```
atmMonitorObject: fileAccess {
  start time: the time of calling for open;
  life time: default long for a file access;
  address: addresses of open and close macro commands;
  sourceType: function for a file access;
  semanticType: data flow;
  constraintBody {
    execution time:
            [AVERAGE, LOWER_B, UPPER_B], or
            a timestamp, or
            subtraction from the starting time/
            AVE_OPEN+EACH*(amount)+AVE_CLOSE;
    inputs: file name/STRING, type of access/READ-WRITE,
            amount of access/[0..FILE_SIZE];
    outputs: amount accessed:
            same as requested;
  }
}
```

Figure 5.1: A monitoring object for highly abstract monitoring

or not the execution takes a reasonable time. An appropriately-sized variable may be used to hold the expected data, so that the variable would overflow if the value of data held were larger than expected. When the overflow occurs, the monitoring only senses a negative value in the holding variable. The negative value is then understood as UPPER_B which can be defined as *1*. That requires only one bit used to record data, but information provided by this type of recording gives a low resolution image. If no lower boundary detection is required, a positive value held in the variable may be interpreted as AVERAGE, which can be held by another one-bit variable. If the lower boundary is of interest to an analyst, one more shift of the data to a smaller-sized variable could be taken, and the result would be either LOWER_B or AVERAGE. As an alternative, the monitored data is simply held by a variable containing the lower-bound value as a base. A two-bit variable is required to hold the data which distinguishes up to four types of information.

For various degrees of resolution, different constraints are chosen to characterize

the monitored feature. As mentioned concerning the execution time, with a possible low resolution the constraint can be taken from the set of LOWER_B, AVERAGE, and UPPER_B. That is, the observed data could be translated into indices of three sets of data. At an immediately higher degree of resolution as defined in Figure 5.1, a timestamp is recorded. The constraint could be the values resulting from the addition of the previous time stamp and an expected interval. At an even higher level (again referring to the definition of the first abstract monitoring object in Figure 5.1), the constraint may include an expected service time and an expected waiting time. If either one is not appropriately matched, then the observed data may be considered significant and consequently be collected. Raising the resolution step by step, the monitor may be able to record its observation more precisely. No matter what degree of resolution is applied, as long as invoked constraints are satisfied with regard to the preferred resolution , the file access should be considered normal.

An alternative of this monitoring, shown in Figures 5.2, 5.3, 5.4 and 5.5, can give more detailed information about the monitored activity and involve three monitoring objects that perform in a sequential manner. A composite monitoring object is created to associate these three objects. For outputs of opening a file, three constraints could be applied: the first for checking the returned file identification, the second for inspecting the file control block created for this access, and the last for detecting anything abnormal about the time for execution. Certainly, there could be more types of constraints involved in monitoring the file opening if a constraint object were created with more concerns.

As defined in the constraint object for observing the reading access, an even higher resolution for watching the execution time at the user interface level serves to observe the performance of each reading. The resulting collection is the sequence of timestamps for the whole reading process, with which a situation such as retrieving some records is slower than retrieving others can be detected.

```
atmMonitorObject:  fileOpen  {
  start time:  the time of calling for open;
  life time:  a return signal;
  address:  address of the open command;
  sourceType:  function for file open;
  semanticType:  data flow;
  constraintBody  {
    inputs:  file name/STRING, type of access/[READ,WRITE]
             amount of access/NATURAL;
    outputs:  file identification:
              a boolean number indicating in [0..19], or
              a number in [0..19], or
              an address of the FCB, or
              part of fields in a FCB, or
              the whole FCB;
    execution time:
              [AVERAGE, LOWER_B, UPPER_B], or
              a timestamp, or
              subtraction from the starting time/AVE_OPEN;
  }
}
```

Figure 5.2: A monitoring object allocated for observing file opening

```
atmMonitorObject:  fileRead  {
  start time:  a return signal from the open command;
  life time:  minimum( rest of the parent's life, or at the time when
              the close function starts );
  address:  address of the read command;
  sourceType:  function for file read;
  semanticType:  data flow;
  constraintBody  {
    execution time:
              [AVERAGE, LOWER_B, UPPER_B], or
              a timestamp, or
              subtraction from the time when open ends, or
              a sequence of timestamps recorded at each reading.
    inputs: file identification:  [0..19];
            amount to be accessed:
              Boolean number indicating in [0..END_OF_FILE] or not, or
              the number of bytes;
            accessing type:
              permitted access;
    outputs: amount accessed:
              same as requested;
  }
}
```

Figure 5.3: A monitoring object allocated for observing file reading

```
atmMonitorObject:  fileClose  {
  start time:  a signal indicating that the reading ends;
  life time:  rest of the parent's life or until close ends;
  address:  address of the close command;
  sourceType:  function for closing a file;
  semanticType:  data flow;
  constraintBody  {
    inputs:  file identification;
    outputs:  ZERO;
    execution time:
              [AVERAGE, LOWER_B, UPPER_B], or
              a timestamp, or
              subtraction from the time when close starts;
  }
}
```

Figure 5.4: A monitoring object allocated for observing a *close* routine

```
cmpMonitorObject:  fileAccess  {
start time:  the time of calling for open;
life time:  set by a default for a file access or
            a preference from a user;
address:  addresses of the opening and closing commands;
sourceType:  file open, file read, file close;
semanticType:  data flow;
constraintBody  {
    execution methods:  sequential;
  }
}
```

Figure 5.5: A composite object at the user interface level

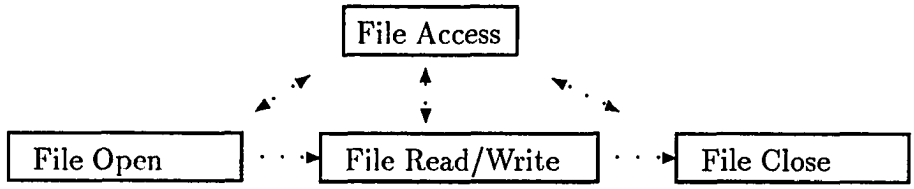


Figure 5.6: The graphical representation of a composite monitoring

The architecture of Figures 5.2, 5.3, 5.4 and 5.5 can be seen more clearly through Figure 5.6, in which the composite monitoring is graphically described. An atomic Monitor object is bound to observe data from a matched type of resources at the desired level of abstraction. For example, the monitoring as defined above for file open, file read, and file close can be carried out by one function-type atomic object but equipped with different constraints since these three monitoring activities are expected to perform sequentially.

5.2.3 Varying Layered Abstraction

When the need for observing details of these operations at the next abstract level arises, the monitoring may decompose each of these actions. As for the *open* operation, it is required to establish a logical connection between the accessing process and the

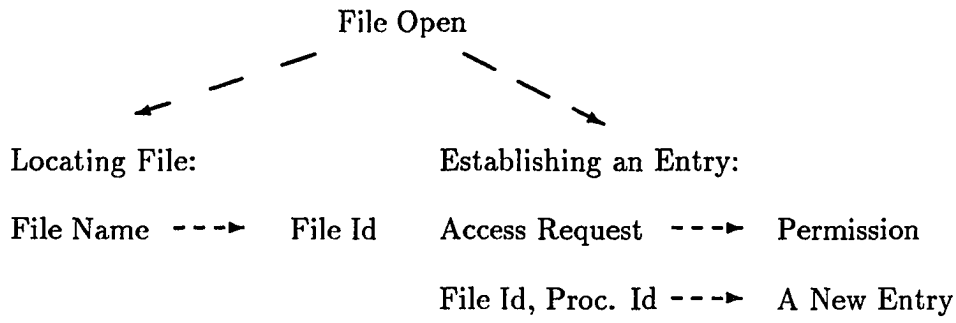


Figure 5.7: The observation at the second level of opening a file

file to be accessed. This operation is accomplished again by a set of lower level operations. The set of operations at the lower level may demand another set of constraint objects to keep track of their activities. Two main monitoring objects and their parent object are organized as shown in Figure 5.7. These two monitoring objects watch for two primary activities composing a file open at its immediately lower level:

- The first task includes locating the file with a given symbolic name and recording information about the current status of the file into appropriate tables. If a file is to be opened for reading, it should already exist, or a failure signal is issued. The monitoring dedicated to this activity should apply constraints to detect the existence of the requested file.

- The next abstract activity to be observed is verifying whether the required access is permitted. If the result of verification is positive, the successive operations are (1) to make a new entry in a private table, which is dedicated to the process requesting the file access, and (2) to fill in appropriate descriptions. The monitoring for this activity may focus on whether a new entry is appropriately created and initialized.

These are the basic activities performed at the level of a file directory system. Similarly, at an equivalent level the actions of file reading and file closing can be detailed

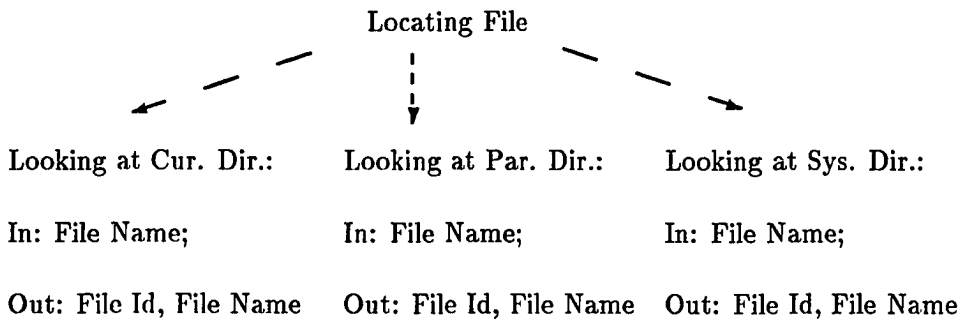


Figure 5.8: The observation at the third level of locating a file

to yield other sets of operations.

Proceeding further, the activities at the second level described above are decomposed into lower-level operations revealing insight into the logical file system. Accordingly, this level demands more observable data. First, the logical organization of files may be of concern to the monitoring, such as types of basic accessing units (either fixed or variable), the manner of organizing records (sequential or linked structures), and logical addresses which could be larger than the size of physical memory but are not sensible at this level. Figure 5.8 shows actions at this third level decomposed from the action of locating the file at the second level. For these actions, a set of monitoring objects should be allocated. However, as previously stated, a sequential monitoring means that one object becomes active after the previous sibling object is deactivated. So, too, when sibling objects are the same type of atomic objects, often only one atomic object needs to be created and to be initialized three times, and at each time a different address for a different directory file is required.

5.2.4 Changing Semantic Concerns

A hierarchical model that abstracts the monitored target can be projected to focus on certain semantic concerns during each moment of monitoring. As specified in the previous chapter, the monitoring may concentrate on collecting different types of

```
atmMonitorObject:  fileAccess  {
  start time:  the time of calling for open;
  life time:  set by a default for a file access or
              a preference from a user;
  address:  macro commands;
  sourceType:  function for file access;
  semanticType:  event-driven;
  constraintBody  {
    type:  SYSTEM_CALL(OPEN, READ, CLOSE);
    execution time;
    state:  file control block, file descriptor, or file directory;
  }
}
```

Figure 5.9: An atomic object for event-driven monitoring

data such as events and data flows. At an abstract level, the monitoring may aim toward components that are geographically associated, or to monitor part of a logical design, or to collect data relating certain resources. Notice that the geographical association may be tangible or intangible; for example, a number of applications may be considered associated when they share the same utilities, such as using the same channels and being adjacent to each other when they are buffered or stored. The illustration of monitoring a file access so far has been addressed to its logical structure by observing data flows. At the level of a user interface, the event-driven monitoring may be conducted by using monitoring objects as defined in Figure 5.9. The detail of a file descriptor should only be needed when observing the underside of a user interface level since it is usually manipulated by routines belonging to a file server.

Another alternative to performing the resource-oriented monitoring is to use a set of variable-type atomic objects. These objects observe the change of each parameter, which is sensible at the level preferred. This way of monitoring may often work effectively, especially when parameters used by each function are passed by address.

```
atmMonitorObject:  fileAccess  {
  start time:  the time of calling for file access;
  life time:  set by a default for a file access or
              a preference from a user;
  address:  address of file identification;
  sourceType:  record for a file access;
  semanticType:  resource allocation;
  constraintBody  {
    visiting intervals;
    the field indicating the position to be accessed;
    change field;
  }
}
```

Figure 5.10: A monitoring object for observing a file control block

5.3 Controlling Data Collection

Based on the established abstract representation, the monitoring may focus on a certain level so that it may ignore details that should be considered at other levels. Thus, data to be observed may largely be reduced during monitoring. Despite this, further control over data collection is provided in AIM. Default reasoning is the main mechanism responsible for such control and follows the procedure which is to be examined hereafter.

Initial Control When there are no explicit demands for the resolution of monitoring, it may often start at a level of maximal abstraction. In the case of monitoring a file access, the monitoring starts by creating only one Monitor object as shown in Figure 5.1. Possibly, at the beginning, only constraints on access time are applied in monitoring. The values bound to each attribute represent the solution of applying a default while the justification of the default is stored in a data base. Constraints that are initially applied are from knowledge derived from qualitative analysis, or expectations based on the initial design of a monitored system, so that certain constraints are not accurate in terms of how the system will actually perform.

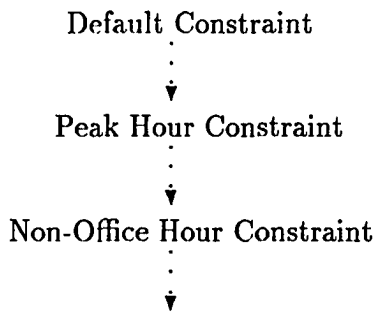


Figure 5.11: A set of ordered predictions for service time

As the monitoring goes on, defaults very likely need to be updated, and alternate constraints may be generated to handle exceptional cases. But the changes will be made only after a sufficiently detailed observation. Hence, it should usually not happen right away at the most abstract level of monitoring. Rather, the controlling strategy is to attempt alternatives first, and if they also fail, then to change the resolution, details, and levels of abstraction. The illustration of this process follows.

Applying Defaults with Order Though initial constraints usually are simple and applicable to general cases, they may eventually be extended to fit into the current monitoring case. The process for evolving defaults will be examined in the next section by using the same example. For explaining this part of the function, assume that changes made on the expectation of the service time for a file access may lead to having a structure such as the one illustrated in Figure 5.11. The order of applying these defaults is sorted through the heuristics listed in Section 4.3.3. With the default hierarchy shown in the figure, the default at the root is first applied when the data is observed. If a violation occurs, the data will not be immediately considered as significant as initially determined. Instead, the data will be filtered through the constraint next to the root. If the results show that the data either does not fit or violates the second constraint, the filtering will consequently attempt the next one until the data is collected or ignored. However, because the modeled default

structure is used, realistically, trials occur on average two or three times. This implies that the hierarchy of constraints is retrained to grow in order to satisfy the criteria for reasoning speed and cost.

Controlling on Resolution In general, the resolution of monitoring is kept as low as possible. After a period of observation, suppose it appears that some accesses are slower than others; then, instead of just recording whether services are slower or normal, monitoring may collect a greater amount of data in order to further distinguish the slow period. Or, finally, it may directly record observed timestamps. The above changes mean the adjustment of physical resolution. As the physical resolution increases, the monitor might still not be able to figure out why, but the clear description of data itself could be obtained.

The monitoring objects illustrated above are anticipated simply to collect data-type description. Above these adjustments, the logical resolution can be addressed at the level of the information-type description. One way to endeavor this high-level description is to associate related observed data items. In the example of a file access, one way to do it is to bind observations of opening and reading files together. The results could yield more detailed information:

- The first situation is that the opening process takes longer whenever the reading process does too. This may reveal that the slow file access is caused by the slow service provided by the file server.

- The second case may be that the file opening takes a normal amount of time while only the reading access takes longer. This may imply memory shortage, or that this file is accessed particularly by many processes. To confirm either possibility, observations of individual readings can be bound together to see the variance of those readings. The results might be two additional situations: one might show individual readings evenly slow, while the other might indicate

that some readings were severely slow but others were basically normal. If the first situation occurs, it is apparently the symptom of too many accesses taking place simultaneously; if the other situation is observed, the conclusion may be a problem of memory shortage or unsuitable buffering methods.

With increased logical resolution, the monitor can distinguish phenomena with better precision. In the three situations detailed above, the monitor may likely accept the first two as exceptional cases and observe the third more closely. The need for figuring out why some readings are slower than others may push the monitoring to engage the knowledge-level description. As the monitoring gets down to the level of physically accessing the data and at the same time tries to maintain the structure of logical records, this requires keeping track of the boundaries of physical records as well as logical records. By increasing logical resolution to the level of accessing logical records or to the level of accessing physical records, collected data may help reveal different causes of the phenomenon; for example, if the slow phenomenon is clustered with a logical-record basis, the question may focus on how logical records are distributed and organized by the file server; if the phenomenon is clustered based on physical records, the source for the anomaly may be physical accessing media such as I/O channels, storages, and others.

Finally, the above observation can be coupled with additional accumulation for mean, variance, and maybe even pattern recognition. Since the second moment can be deduced from the mean value, the high moments are not suitable to be calculated on-line. On the other hand, tracking a mean value can be done indirectly, e.g., to keep a counter and an accumulating record for a mean visiting interval. Nevertheless, which way is more suitable depends on how an appropriate buffering method is applied.

Switching Levels As implied in the foregoing demonstration, the monitoring may go down to lower levels from the user interface level to observe activities inside the file

server. At each lower level, there is also more than one way to perform the monitoring, including shifting semantic concerns from level to level. A typical observation is to focus on the utilization of involved resources. In this case, i.e., a file access, the monitor may often watch for a related file descriptor. Such monitoring can be carried out by the monitoring object as the one shown in Figure 5.10. Defined in that monitoring object, one observation is to monitor the change of the current logical address and to record the mean interval of visiting the field that stores a logic address. If, during the execution of a reading operation the mean interval of visiting appears lower, or, in other words, a high percentage of data is collected because the data is near the lower boundary of a predicted mean value; then one may conclude that accessing memory takes excessive time. Since accessing a logical data unit involves a two-step process—mapping address and accessing memory—distinguishing the part of the process which causes the problem may demand the monitoring to observe changes of the current mapped physical address. No matter whether the resulting mean value is normal or still low, the information may imply that the problem may be too-frequent paging because of a lack of memory, or that the process of mapping the address is slow, probably due to a slow cpu service.

If a deeper-level monitoring is requested and required monitoring objects at that level are not available, the creation of a composite object will be conducted through the set of operations specified in Section 4.2.2. Sometimes a monitoring object bound to a composite monitoring may contain attributes unnecessary for the joined monitoring. For example, since three monitoring objects for observing the "file open," "file read/write" and "file close" satisfy the criteria of the operation *horiz-concatenation*, they are bound with a sibling relationship as defined in Section 4.2.2. Then, a monitoring object, initially allocated for monitoring the performance of the routine of opening a file, may be used to monitor subsequent activities. Thus, probably only part of the constraints actively participate in the rest of the observation. If the con-

straint for a file name is no longer needed in observing readings, this constraint can be deactivated through the operation called *projection.*

Note that levels of representation are also reducible. If the data collection proceeds in a smooth manner, the monitoring may go back one level by following the principle of minimizing data collection, and monitoring objects performed at the lower level may no longer need to exist.

5.4 Refining while Monitoring

Defaults evolve on the way to becoming more suitable to a monitored system. At the time an atomic monitoring object is created, it may have only a single default. It may likely often conflict with phenomena observed, or it may be unable to characterize a target. Accordingly, a possible evolution from a single default prediction to an ordered set might lead the default structure as shown in Figure 5.11.

To exhibit the evolutionary process of AIM, one may assume that an intention of monitoring file access is to learn why sometimes the access is slower than predicted. This requires first finding out whether the slow access happens regularly or randomly. If it is a regular phenomenon, then one may want to see during which periods it appears; otherwise, the monitor likely needs to provide more data regarding the parts which slow down the whole system.

As discussed previously, the resulting model is capable of evolving constraints in terms of the performance of monitoring. By continuing the example used so far in this chapter, the rest of this section will show how the refining is conducted. Previously specified types of refining fall into three categories. The illustration will demonstrate the three cases accordingly.

5.4.1 Accepting Exceptions

Continuing the discussion of monitoring the service time of a file access given in the previous section, suppose that the first two situations lead to this conclusion: during the peak hours, 9 am to 12 pm, a file access takes about 30% longer than normal; then this new estimate is backed up to the initial default. The result is that during other hours, whether the data can be considered significant can still be measured by the initial default, but during 9 am to 12 pm the significance of observed data is instead determined through the second estimate. Later on, some other symptoms may also be detected. For example, one may see that the amount of significant data is substantially increasing on Monday and Friday. After observation, it is found that the symptom appears regularly and does not cause other incidents in the monitored system. Furthermore, by looking into the details of collected data during these days, if one notices that most of the data collected on Monday reach or come near the upper boundary—implying the service time is longer than an expected average—and by contrast one finds that the significant data observed on Friday are close to the lower boundary, to allow for these special cases, the default hierarchy is expanded to be something like that in Figure 5.12. Cases on Monday and Friday have no direct influence on each other; therefore, they are not ranked against each other. By comparison, the special estimate for morning hours may overlap with these two cases, so that together they are associated with the relationship of parent-child. The issue of which case should be the parent case depends upon the heuristics defined in Section 4.3.3. If peak hours impact business more than Monday and Friday office hours do, or peak hours have more coverage than the Monday and Friday hours, the relationship should be established as shown in Figure 5.12.

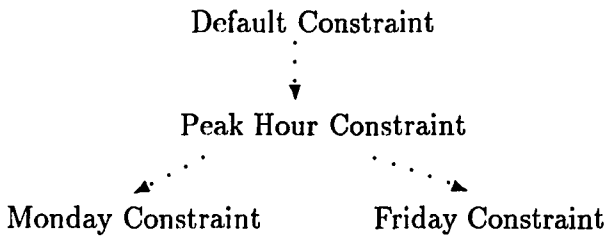


Figure 5.12: A default hierarchy compromising the estimates on special weekdays

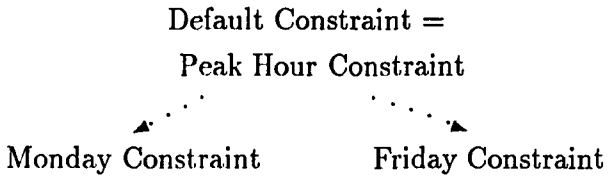


Figure 5.13: The default hierarchy after the peak hour default replaces the initial one

5.4.2 Updating Classification of Defaults

To show the default adjustment, imagine that the monitored system gets busier in the afternoon due to certain reasons. Consequently, peak hours are extended to include the times from 1:30 pm to 4:30 pm, and the afternoon peak hours also make the service time almost as long as that needed during the morning peak hours. This leads to the measuring constraint for the service time during peak hours being referred to more often than the initial default for the expected service time. By applying the second demoting criteria (see Section 4.4.3), the refining process is triggered to replace the initial default with the constraint for peak hours. The resulting default hierarchy becomes the one depicted in Figure 5.13. The new exception of service time now refers to lunch break, early morning, and later afternoon, during which the service time is shorter than the average, i.e., the default.

It is, of course, reasonable to expect a default to be refined sometimes instead of being replaced. This happens particularly when, for example, the monitored system, is upgraded by adding additional parts such as extra memory, disks, and/or faster

CPUs. To keep using the old default may cause too much data to be collected because more data is generated with intervals near the lower boundary and distributed evenly during weekdays; if no lower boundary is set, there may be little data collected. In response to this situation, monitoring looks into the phenomenon in more detail. After the observation continues for a certain period, (the length of which depends on a default that is possibly adjusted at a later time), but there is still too little or too much meaningful information in terms of constraints applied, while the performance is nonetheless generally smooth and stable, then either tightening or relaxing the constraints should be suggested. Thus, the default is refined but not replaced.

5.4.3 Refining Controlling Methods and Refining Strategy

Some applications of adjusting triggering criteria and refining strategy are examined in this section. One of the important refinements in these two aspects is to adjust the weight of each heuristic. As usual, a constraint more applicable to the observed phenomena is favorable. However, this should not always happen. While a monitored system appears stable regarding its performance in different periods, the monitoring may need to switch its attention to different perspectives of the system's performance, such as to observe behavior of batch processes or interactive processes. Thus, the service time should be measured through different defaults in accordance with different types of service. To realize such a change, the weight of Heuristics 2 and 3 are gradually reduced whereas the weight on Heuristics 1 and 4 are increased. With the emphasis on selected heuristics, the defaults may be organized with the order as described in Figure 5.14 in the monitoring if interactive computing activities are more frequent and more troublesome. This means that, upon violation of the general default, constraints to be applied in the next trial are those specified for a batch process or an interactive process.

Moreover, another mechanism for the refining process is to constrain the trend

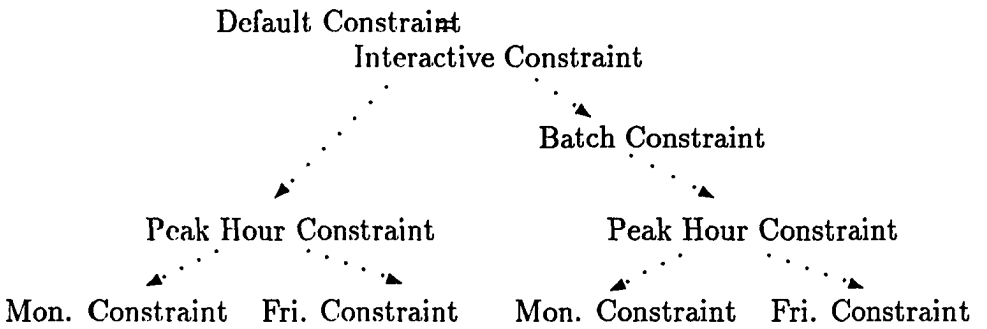


Figure 5.14: The default hierarchy considering batch and interactive processes

that may overstress symptoms and may finally lead constraints, which characterize computing activities, to slide away. As specified in Section 4.4.3, this is achieved by adjusting the pace of increasing or decreasing the weight of each heuristic. In the above example, the default hierarchy displayed in Figure 5.12 may be appropriate in controlling the amount of significant data while the monitoring is still able to catch most significant data. Hence, such defaults as a set can be assigned a high weight in terms of Heuristic 4. Nevertheless, allowing the weight to increase without resistance may cause a problem: the longer a default or a set of defaults has been used, the more likely the other defaults will have been overlooked. Therefore, the necessary balance on the total weight of a default is maintained.

Finally, it is desirable to refine the semantics of constraints. For example, constraints on access time for distribution of hours and types of jobs can eventually be combined to form more effective defaults if the observation shows that these two sets of defaults are often used together in making a judgement. Refining in this direction may produce a better knowledge chunking and yield combined concepts.

Chapter 6

A Prototype Implementation

A prototype of AIM, which illustrates basic aspects of the researched approach, has been implemented. The motivation of this implementation is twofold:

- To demonstrate that the specified model for an intelligent approach is operational.
- To provide a testbed for evaluating the relevant mechanisms which have been integrated into the resulting model.

6.1 Orientation of Implementation

Prior to dealing with details of the prototype, a general introduction is given to elaborate on the intent, make-up, and limitations of this implementation.

6.1.1 Statement of Objectives

To fulfill the stated objectives of this research, certain noteworthy capabilities designed in AIM need to be tested and evaluated through implementation. The following capabilities are prototyped for this purpose:

- The model can significantly reduce the amount of data to be collected, while the performance of a monitored system can be better understood.

- The model may behave with considerable flexibility, especially in monitoring a dynamic system, as compared to those monitoring tools which depend on procedure methods.

- The model relies on declarative knowledge to improve the performance of monitoring during on-line monitoring. This means that monitoring may efficiently depict characteristics of a monitored system while the behavior of a monitored system changes.

Moreover, an important aspect of prototyping AIM is to experience that, in comparison with conventional approaches, the above improvements are possible without disturbing a monitored system. Consequently, the implementation intends to show that these capabilities outweigh possible disadvantages, particularly regarding additional computation.

6.1.2 Implementation Composition

In accordance with the specified model, the prototype of AIM also mainly consists of three parts, namely, representational mechanisms, controlling mechanisms, and refining mechanisms. In order to minimize needs of the computing resources that pertain to a monitored system, the prototype distributes those functions to other machines to be carried out if they hold to the following conditions:

- The functions require less input and generate output that is either smaller in amount or has no need of being returned. This means that one principle of distribution is to minimize the amount of information transfer caused by running those functions off-line.

- The functions are allowed more time to produce results.

Parts of the prototypical model that function remotely are those facilities for displaying monitored data and refining constraints, as well as the main part of the knowledge base that includes both application-dependent knowledge and controlling knowledge. Consequently, besides sending monitored data to remote machines, some additional data flows exist between the buffers for storing on-line controlling knowledge and the knowledge base which backs up all needed knowledge. Updating the knowledge in the buffers is required when the corresponding part of the information in the knowledge base is changed.

The implementation concentrates on revealing underlying data structures and details of algorithms. In implementing structures such as tables, hierarchies and records, the intention is to simplify the parts that may have to reside inside a monitored system and make them easy to use so that the actions of searching and accessing need only minor computation. Algorithms are designed to be as generic as possible, and are heavily dependent on declarative knowledge so that flexibility is strengthened and complexity is reduced.

While still offering an analytical display of the ongoing monitoring, concern for the cost of running a monitoring system is motivation for finding a method which can be used to pass collected data quickly to other machines for further examination. Inasmuch as data are observed through a preliminary process and then are picked up by remote machines, the amount of data passed to other machines is greatly reduced. An exhibition of collected data is shown on multiple windows based on certain classifications. This makes further extension of analysis on the performance of monitoring possible. The current display facilities are accommodated by use of network sockets and X11 windows. Thus an integration of these facilities into some sophisticated graphical package would be easily realized if such a need arose.

The prototypical monitoring system functions on top of an event and data generator which is described in detail in appendix C. By means of simulation, the experiment overcomes the problem of having no suitable software packages to be accessed which may provide a computing case as shown in the example examined in Chapter 5. In addition, by use of a simulation generator, the effectiveness of experiments can be reached; the generator provides typical data for representing data samples and various patterns for each level of abstraction over a reasonably long period of time.

6.1.3 Limitations

While the intention is to demonstrate major functions specified in earlier chapters, the depth and width of the implemented model are naturally smaller than the model previously designed. Keeping this implementation aimed toward the above indicated objective, the development of the prototype has cut off details which could be necessary for a monitoring system in practice, and is not concerned with the following:

- It does not provide any deep reasoning mechanisms, and it does not have an explanation system to offer further analysis on collected data.
- It does not support query mechanisms except a few interrupting commands, so human intervention may be accepted during monitoring.
- It does not have sophisticated methods for maintaining a large-scale knowledge base.
- It simplifies possible cases which may be seen in monitoring a real system, into fewer categories.

Such limitations will be indicated in each section where related implementation is discussed.

6.2 Structure for Storing Constraints

Constraints are organized in terms of their semantics and are associated with rules from which constraints are derived. Constraints are first grouped to describe a monitored activity; then grouped constraints are classified into sets, each of which is dedicated to one kind of resource. Constraints are stored on the basis of what types of targets they intend to characterize. Similar to the description given in Section 4.2.1, under the category of physical resources, could be those belonging to hardware resources—memory units, channels, processors, and buses; and those belonging to software resources—routines, statements, processes, stacks, queues, variables, files, and processes. Logical resources also include policies and algorithms, dealing with accessing, scheduling processes, paging, swapping, mapping, and some others. In the category of functional resources there are groups of constraints capable of describing functional aspects of an entire monitored activity or part of the activity. In supporting default reasoning, often several constraints or several groups of constraints may be used to characterize a target or the attributes of a target. Those related constraints and groups of constraints are then organized into a hierarchy based on the weight tagged to each constraint or group of constraints. These weights are determined by evaluation based on heuristic principles as defined in Section 4.3.3. For the reason of achieving optimal performance, only a small part of the constraints are stored in buffers for quick access by on-line monitoring functions. In the prototype, such a buffer is organized into a two-dimensional matrix as shown in Figure 6.1. Each entry in the matrix stores a default constraint or a default group of constraints and connects to a tree-type structure that holds various constraints which may characterize similar features but which are assigned various degrees of priority. The matrix is periodically updated if its corresponding part in the knowledge base is changed. The updating frequency depends on how much the constraints are changed or whether the maximum period is reached.

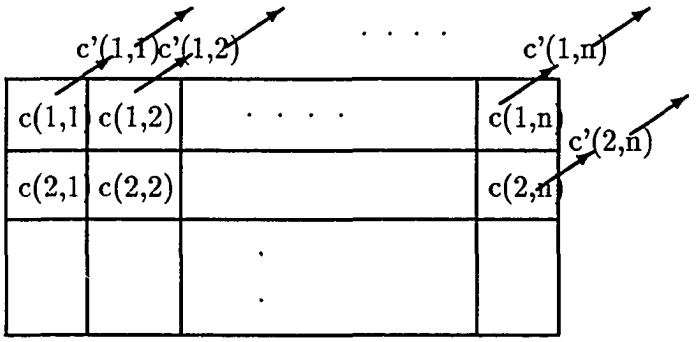


Figure 6.1: The data structure storing constraints

6.3 Generation of Objects

There are two basic types of monitoring objects available in the prototype: atomic monitoring objects and composite monitoring objects. Treating the above data storage as the fundamental basis of controlling knowledge, an atomic object can be generated by allocating it a row in the matrix. The entries in the row conjoin to characterize an activity or a feature which is intended to be monitored. Creating an atomic object also requires choosing related constraints which are bound to the constraint at each entry of the matrix. Together they form a hierarchical structure so that controlling data collection with the default reasoning method may operate with an appropriate basis. The virtual accessing method—to make the limitation of buffer size transparent to the on-line controller—is not implemented even though it is desirable to do so, because it would add too much to the cost of running a monitoring tool. Rather, it is preferable to have a small knowledge base stored on-line for supporting the default reasoning. The positive aspect of this design is that it may effectively limit the cost from such control; the negative aspect is that it may further weaken the soundness of default reasoning. However, depending on a small knowledge base may be permissible in the following situations:

- The knowledge base can be presumably be updated at any time. Hence, when the controlling mechanisms based on available knowledge do not give satisfactory performance, the off-line refiner then makes some improvements on the knowledge instead of keeping the knowledge base unaffordably large to cover all possible cases.

- Unlike diagnosis, the accuracy of monitoring is desirable but not critical; on the other hand, a heavy disturbance may drastically damage the quality of monitoring, which, by contrast, is not an issue in most diagnostic cases.

At the time of generating a monitoring object, it is assumed that related constraints are sorted in a tree structure. Then such a tree is moved to the on-line buffer with its root allocated at an entry of the matrix. The rest of the tree is stored along the third dimension with the same partial order. By the principle of default reasoning, the root constraint is always tried first. If the default constraint leads to too much or too little data collected, then other constraints are tried in order. The following piece of code displays a high-level routine that accommodates the function of generating an object. The routine is called when a request for generating an atomic object is issued. Attributes of an atomic object are initialized with defaults. Types of attributes in an atomic object are predefined. Figure 6.3 displays the additional code for initializing a newly generated atomic object.

Composite objects are composed of atomic objects and/or composite objects through operations defined in Section 4.2.2, such as conjunction, unification, and unary operations. This process is carried out in two ways: user-machine interaction, and dynamic generation through the monitoring controller. The prototype supports the process of composing composite objects through reading definitions from a data dictionary, including a standard input file. The grammar used for defining a composite object is shown as follows:

```
struct atomObject
       *atomic_object(typeObj, prinPtr, prefPtr)
Types typeObj;
Strings *prinPtr;
Strings *prefPtr;
{
  obj = (* coreObject) malloc(sizeof(coreObject);
  if (obj == NULL) return ((struct coreObject *) NULL);

  While ( attrbPtr ) {
      look_for_constraint(attrbPtr, prinPtr, prefPtr);
      add_to_object(obj->attrbAry, attrbPtr);
  }
  return (obj);
}
```

Figure 6.2: The routine generating atomic objects

```
  :
obj = compose_atomic_object(STACK, prinAry, prefAry);
if (obj != NULL) {
  obj->parent = parentObj;
  obj->startTime = max(parentObj->startTime, current_time());
  obj->startTime = min(parentObj->lifeTime,
                       parentObj->lifeTime -
                       (current_time()-parentObj->startTime));
  obj->probe = (long) address(PROBE_STACK, &stack);
}
```

Figure 6.3: Initializing a monitoring object

$compObj$ :: $(< compObj > | < atomObj >)OP(< compObj > | < atomObj >)^*$

OP:: CANJ | UNIF | PROJECT | SELECT

CANJ:: VERT_CONC | HORIZ_CONC

UNIF:: NEST | MERGE

All atomic objects to be nested must already exist. All nested composite objects may already exist or may be dynamically generated. As long as a class to which each nested composite object belongs is predefined or can be found in a data dictionary, a composite object of this class can then be created. Since definitions of an object class in a data dictionary are continually updated, the types of composite objects are redefinable.

The corresponding code for generating a composite object according to the above is listed in Figure 6.4, with explanation given below. Under the category of conjunction, only the operation adding two objects is implemented; within the category of unification, only the merge operation is implemented. Projection is done by initializing some of the attributes to corresponding default values and setting all other attributes to a special value such as zero.

Once a composite object is generated, the logical structure of the composite object is stored in a table with fields indicating the head part and indices to participating atomic and composite objects. All tables are linked together and are maintained again by the Monitoring Controller. Figure 6.5 shows such a table that stands for a composite object. Such tables finally connect to the matrix that stores atomic objects. A composite object may associate with more than one row, or perhaps it may link several rows together to characterize a monitored target. Due to this design, which does not allow a composite object to take over parts of the rows in the matrix but only indexes to those rows, the method leads to atomic objects being shared among

```
get_token(strPtr, token);
if (strcmp(token, ''COMP'') != 0) {
  return (KEYWORD);
}
if ( (next_char(strPtr) != ':')  &&
     (next_char(strPtr) != ':')  )  {
  return (DELIMITER);
}
get_token(strPtr, token);
if (object_type(token) == FAILED) {
  return (ILLEGAL_SYMBOL);
}
if (get_token(line, token)) {
  if (exist(token) == NULL) {
    return (NON_EXIST);
  }
}

while ( get_line(strPtr, line) != NULL ) {
  if (get_token(line)) {
    obj->proxy.type = convert_to_type(token);
  }
  while ( get_token(line, token) != '>' ) {
    switch (obj->proxy.type) {
    case CONJ:
      connect_an_object(token);
      break;
    case UNIF:
      add_object_to(token);
      break;
    case UNAR:
      initial_selected_attrib(token);
      set_other_attrib_zero(token);
      break;
    default:
      return(ILLEGAL_TOKEN);
    }
  }
}
```

Figure 6.4: The code for generating a composite object

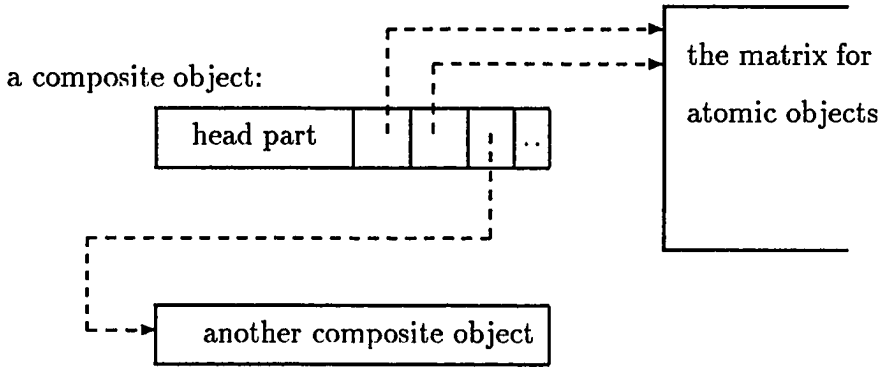


Figure 6.5: The table for holding a composite object

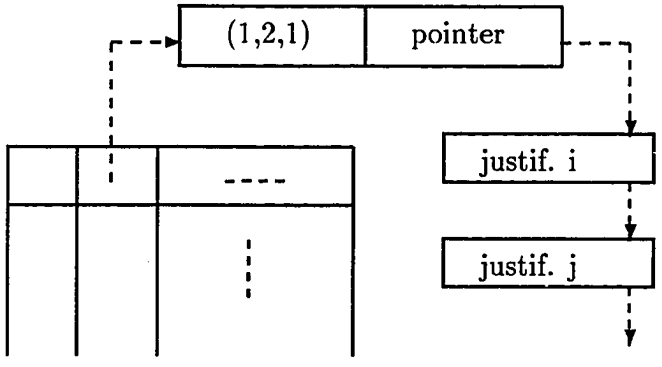


Figure 6.6: The mapping structure for a constraint and its justifications

composite objects.

6.4 Rule Structures

Constraints are derived from rules that provide the justification for the constraints; together they constitute a knowledge base for default reasoning. In consequence, constraints must be associated with rules in order to provide users with deeper explanations and to assist further in refining constraints by providing updated rules. Association between rules and constraints are realized by mapping through the method as shown in Figure 6.6. Each constraint in the matrix is identified by its location,

Id	caseDesc	pointers to justification rules	solution

Figure 6.7: The format of a rule

Symbolic Name	SetId(1)	SetId(2)	· · · ·	SetId(n)

Figure 6.8: The format of a rule head

namely its row and column indices. This identification is used as a pointer to bind to a set of justifications. The prototype simplifies the index structure by limiting each predicate to be dedicated to one constraint. If one predicate yields (or, together with other predicates, conducts) more than one constraint, the predicate is repeatedly stored as many times as the number of constraints with which it is involved. The process prevents the pointing structure of justifications from becoming too complicated. But it should be recognized that some justifications, particularly those pertaining to domain-independent knowledge, should not be repeatedly stored, because this practice may lead to considerable waste of storage. Fortunately, this type of knowledge is usually stored elsewhere, so more space is taken from remote systems than from inside a monitored system. This results in little disturbance to the monitored system.

A rule supporting default reasoning consists of four parts: rule identification, condition, solution, and a pointer to related predicates providing justification. Furthermore, rules are grouped into sets to benefit more complicated default reasoning. The format of a rule is exhibited in Figure 6.7. The detail of a rule identification is shown in Figure 6.8. The rule identification includes the name and classification of rules. The classification mark is needed for efficient organization. It does not help significantly in this prototypical model, but it may greatly improve efficiency, which is essential for an intelligent system to have a practical value.

So far the description of the prototype has presented the structure of constraints

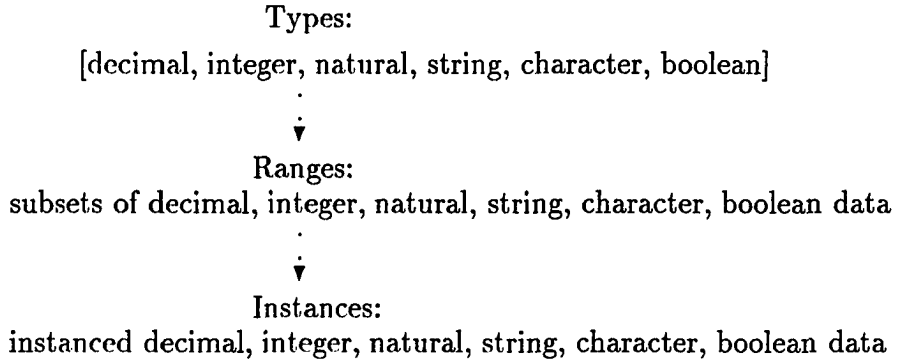


Figure 6.9: A resolution hierarchy shared by constraint objects

and the underlying knowledge. The next step is to implement the classification of constraints. Again, note that there are two types of hierarchies built upon the constraint objects: a semantic hierarchy, and a layered abstract model for simulating a monitored system. Virtual objects are often used to combine monitoring objects that represent different abstractions of a monitored target; they no longer exist after the corresponding monitoring is done. There is no necessity for fixed storage for those virtual objects. Neither do they have classes from which they are generated. Instead, they are generated based on the logical relationship of a monitored system. A hierarchy represented by a composite object often does not have inheritance to talk about, but it does have most of the other features of object orientation, such as encapsulation, abstraction and overloading.

Figure 6.9 shows a skeletal organization of resolution hierarchies. The constraint object at the top level is the one that contains the loosest constraints. The term *loosest* refers to those constraints having the least resolution and often requiring the least computing effort. It can be seen in Figure 6.9 that the top-level objects include highly basic attributes and omit specific attributes that might vary from case to case.

6.5 Controlling Functions

The implementation of controlling mechanisms follows the steps listed below.

1. Apply default constraints which should be assigned in terms of qualitative analysis. There are usually two ways to do this: to have a group of experts with experience in similar systems assign defaults, or simply to take the mean values from the initial observation as defaults.

2. If the data collected is too little or too much because of the guidance of a criterion stored in the shared memory which is treated as part of the knowledge base, then select another constraint to measure the observed data. The shared memory is used as a buffer for controlling rules and can be updated autonomously through the refiner or manually by a user.

3. Repeat the above steps until no more alternatives are available; if there are no more alternatives, then proceed to the next step.

4. If observation seems to be abnormal, then increase the resolution, both physical and logical. This step includes allocation of more room for recording data and managing more statistical recording for later analysis.

5. If the observed data does not bring sufficient confidence to the ongoing monitored activities, or if the observed phenomena require more comprehensive descriptions of monitored features, apply more constraints until precluded by some criteria or until no appropriate constraints are available.

6. If an observation consistently shows an anomaly, go down one level and repeat from Step 1.

The pseudo code for the above algorithm is given in Figure 6.10; the algorithm is designed to function within a hierarchical default structure. Its success also relies on

```
root = choosing_semantic_tree(type);
while (root->attrib[ii] != NULL) {
  while (root->attrib[ii] == user_pref[jj++]) {
    over_write_attrib[ii++] = user_pref[jj-1];
  }
}
for ( EVER ) {
  apply_constraint_obj(root, period);
  while ((results = check_criterion(root->statistics,
          criterion[kk++])) != NULL) {
    adjust(root, results);
  }
  if ((id = more_detail_req(root->statistics, user_pref)) != NULL) {
    root = root->child[id];
  }
  else if ((id = more_abst_req(root->statistics, user_pref)) != NULL) {
    root = root->parent[id];
  }
}
```

Figure 6.10: Pseudo code for default reasoning

whether the moderate size of a hierarchy can be maintained since a large hierarchy may cause the default reasoning to become too costly. This is accomplished essentially by making obsolete those constraints that are less frequently used (see the next section). Once such a hierarchy is well maintained, applying constraints often starts from the root of the hierarchy to ensure that default constraints are applied first. However, sometimes it may be more efficient to try alternate constraints first if other heuristic principles become more important.

6.6 Refining Functions

Three basic refining methods are implemented. The first evolves case hierarchies, the second adjusts defaults, and the third adapts parts of meta knowledge. The routine shown in Figure 6.11 is responsible for adjusting steps of increase for boundaries of

constraints. This function assists in the third part of refining, which is somewhat meta-flavored refining. The second refining method is to adjust defaults and make them more accurate, since initial default values result from qualitative experience which is often based purely on the understanding of a typical system among similar types. There are several similar routines developed for this type of refining, but they vary on how defaults are to be modified. A typical routine for this function is displayed in Figure 6.12; it performs the adjustment of constraints for expected intervals.

The last part of the implemented refining functions to be examined is the realization of the extension of default cases. Cases are sorted according to several concerns. Increasing the weight of a case results in the case moving up along the hierarchy, and the converse is also true. Every comparison between two case weights may suggest changes in those weights, especially if positions in a hierarchy are not consistent with their weights. The pseudo code in Figure 6.13 adjusts the order of nodes whose priority becomes lower. Remember that the deletion of a constraint object means that the depicted feature(s) is no longer significant with respect to its characterizing strength on the performance of a monitored system. If the feature later appears, it will be considered an exceptional case. Ideally, the number of deletions of each case should be recorded. The recording may be used to resist possible deletion of similar cases that perhaps recur, so that some phenomena which might occur periodically would finally be recognized by the monitor. However, this function is not accommodated in this prototype, and it is doubtful whether this type of exception can be effectively controlled without extensive research in areas such as time series forecasting and pattern recognition.

```
set_Delta(maxLorS,minLorS)
int maxLorS, minLorS;
{
    FILE *deltaBase;

    get_Delta();
    if (maxLorS == 1) {
        maxDelta += STEP;
    }
    else {
      if (maxDelta-STEP > minDelta)
        {maxDelta -= STEP;}
      else
        {maxDelta = minDelta;}
      }
    if (minLorS == 1) {
      if (minDelta+STEP < maxDelta)
        {minDelta += STEP;}
      else
        {minDelta = maxDelta;}
      }
    else {
      if (minDelta - STEP > 0)
        {minDelta -= STEP;}
      else
        {minDelta = 0;}
      }
    if ((deltaBase=fopen("deltaB","w+"))==NULL) {
      printf("err:open deltaB");
      return;
    }
    fprintf(deltaBase,"%d %d",maxDelta,minDelta);
    fclose(deltaBase);
}
```

Figure 6.11: The code for adjusting the pace of adjustment

```
adjust_t(whichOne)
int whichOne;
{
    if (whichOne == 0) {
        t1[0]+=LITTLE;
        t1[1]+=LITTLE;
        t1[4]+=LITTLE;
        t1[5]+=LITTLE;
    }
    else if (whichOne == 1) {
        t1[0]-=LITTLE;
        t1[1]-=LITTLE;
        t1[4]-=LITTLE;
        t1[5]-=LITTLE;
    }
}
```

Figure 6.12: The implementation for adjusting expected intervals

```
if (the node is a leaf)
then
    {do nothing}
else if (the node has child objects)
then {
    do
    {switch the node with the one that has
      the largest weight at the next lower level}
    until ( the switching cannot go further )
}
```

Figure 6.13: The algorithm for deleting an exception

6.7 Acquisition Preprocessor

The prototype provides an analyst with a minimal accessing medium for entering necessary commands and multi-windows for displaying observable data. The displaying functions are allocated to remote machines to minimize the interference to a monitored system. Data displayed through multiple windows assists in a meaningful exhibition. Which window displays which data is determined by a classification characterized by applied constraints. Since constraints stand for conclusions of rules, which may be designed by an analyst, the analyst may utilize the multi-window displaying medium to make an analysis on the collected data become easier. Furthermore, each group of data is identified with different tags, including type, range, clock, levels of abstraction and others. The code in Figure 6.14 shows main functions of this part of implementation. The calls for using sockets to pass data from a monitored system to remote machines where the display and the analysis are carried on. In order to reduce the cost of passing data, it does not ask for a "handshake agreement" each time a data packet is passed. The code for establishing a multi-window environment is implemented with the X11 widget set, but it is not examined here since the work done on this part is not emphasized in this research. The statements of "sent_to_lilac(dataMsg)" and "sent_to_sleet(dataMsg)" are calls for functions that pass collected data to buffers in different remote machines where an analysis may be conducted without interference to a monitored system.

6.8 Summary and Discussion

The functions described above have been implemented and tested based on the targets emulated by the simulating generator. Figure 6.15 shows the general structure of this prototype. A brief examination of this chart may serve as a short summary of how each main function operates.

```
:
write_to_file(gid, tm, data)
int gid;
long tm;
double data;
{
    char dataMsg[SECTOR];
    FILE *analy;
    switch (gid) {
    case 0:
        sprintf(dataMsg, "Fst:%ld; %f", tm, data);
        sent_to_lilac(dataMsg);
        break;
    case 1:
        sprintf(dataMsg, "Second:  Fst:%ld; %f",tm,data);
        sent_to_sleet(dataMsg);
        break;
    case 2:
        :
    }
}


    :
  int sd;
  struct sockaddr_in name, from;
  char   buf[1024];
  int    cc, lenfrom;

  sd = socket (AF_INET,SOCK_STREAM,0);
  name.sin_family = AF_INET;
  name.sin_addr.s_addr = htonl(INADDR_ANY);
  name.sin_port = htons(12345);
  bind( sd, &name, sizeof(name) );
  lenfrom = sizeof(from);
  for (;;) {
      cc=recvfrom(sd,buf,sizeof(buf),0,&from,&lenfrom);
      buf[cc] = NULL;
      printf("First Group:  %s", buf);
    :
  }
```

Figure 6.14: The routines for storing collected data at a client site

As indicated in the figure, the data generated from the simulator varies by arrival intervals, types, and ranges, seemingly as well as from different abstract levels of a computer system. Constraints and criteria—for controlling the monitoring and for triggering the refining—are stored in tables and files with structures as described earlier in this chapter. The monitoring part collects data in terms of significance which is determined by constraints featuring arrival intervals and values of data, types of data, the percentage of each type of data collected, and levels of abstraction. If too much or too little is collected when these constraints are applied, more features, higher resolution, or lower-level monitoring may be conducted in a specified order. The adjustment of constraints and the strategy of monitoring is processed by the refiner.

The prototypical model is running on Sun stations, the bottom of which is hidden from the reach of this prototype and is supported by different file servers. Consequently, the accuracy of estimation of the performance of such an intelligent monitor is dependent on the method that measures gain and loss due to intelligent monitoring by running the prototype with intelligent functions and then running it without those functions. To conduct this experiment, the prototype makes the controlling part, the part for displaying monitored data, and the refining part loosely connected to each other. These main components do not share variables and invoked functions. The communication among them is through data packets. The connection of these major components is depicted in Figure 6.16. The information passing between on-line data collection and multi-window displaying facilities as well as the refining process is accomplished through sockets and files while intermediate storage is supplied. The communication between the parts carrying on-line tasks is supported by interprocess communicating facilities with which the prototype uses shared memory and message queues. The results obtained from experiments with the above method are summarized below.

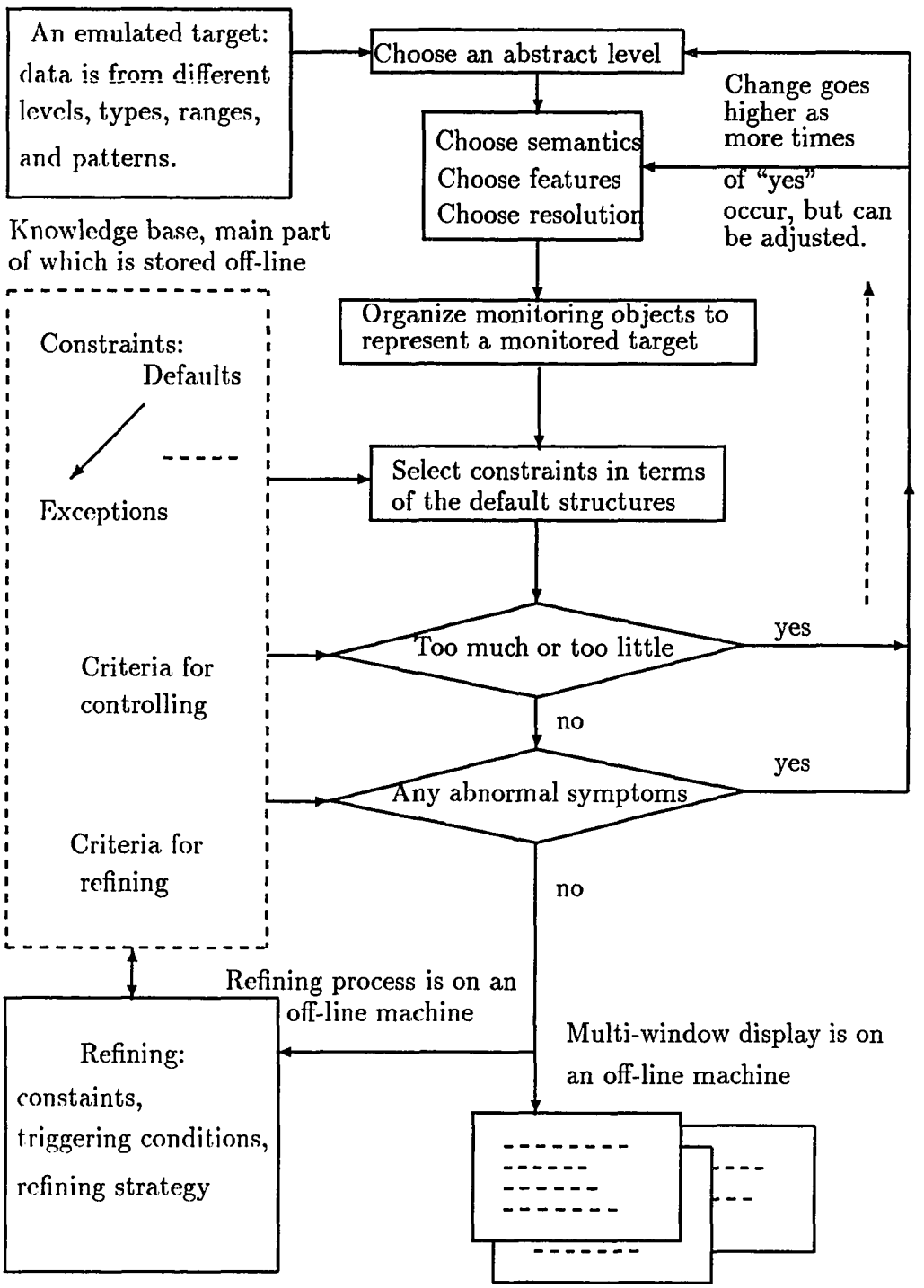


Figure 6.15: The functional structure of the prototype

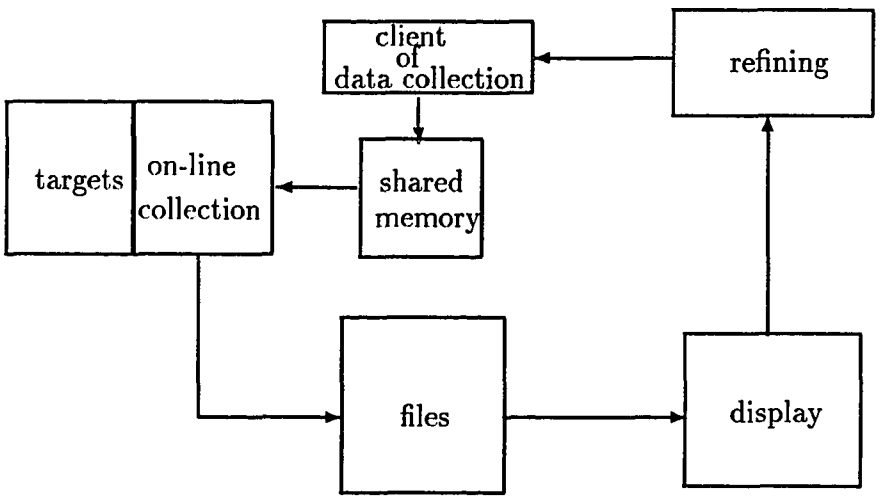


Figure 6.16: The connection of main functions

1. By applying alternate constraints, the amount of data that is considered significant is only around 10% of the total data generated during the period of observation. Some observed data is sampled in Figure 6.17. The collected data belong to three types at each level, and there are three levels in all. Observed data is filtered against defaults and possibly collected if data touches either the upper boundary or the lower boundary set for arriving intervals or for values of data. For the data considered abnormal only because of its arriving rate, timestamps are recorded but not the values of data. The normal data only contributes to the counters, but the data itself is drained.

2. Since the on-line control of data collection does not perform any reasoning in the sense that the filtering mechanism does not refer to rules but simply applies constraints to measure observed data, when the filtering process is temporarily suspended, the speed does not show any difference in the precision of microseconds. Figure 2 shows the data dumped from the log files when the prototype runs in both ways. With the similar arriving patterns, the distance of every two events monitored with no filtering is, on average, close to the one with filtering.

```
T:701563999420000 rdn:2.024395 (ignored)
                  rdn:0.820328 (others)
T:701563999480000 rdn:2.048350 (ignored)
                  rdn:1.960419 (others)
T:701563999520000 rdn:2.138150 (ignored)
T:701563999560000 rdn:2.127692 (delayed)
                  rdn:0.792252 (others)
                  rdn:1.374084 (others)
T:701563999620000 rdn:2.015299
T:701563999660000 rdn:2.103196 (delayed)
                  rdn:1.435318 (others)
                  rdn:1.746552 (others)
T:701563999720000 rdn:2.041639 (ignored)
T:701563999740000 rdn:2.060537 (ignored)
T:701563999780000 rdn:2.161169 (ignored)
                  rdn:1.042908 (others)
T:701563999820000 rdn:2.016051
                  rdn:1.102798 (others)
T:701563999860000 rdn:2.074140 (ignored)
                  rdn:0.578827 (others)
T:701563999920000 rdn:2.382187
                  rdn:1.578506 (others)
                  rdn:0.328186 (others)
T:701563999980000 rdn:2.086273 (ignored)
T:70156400020000  rdn:2.257764
T:70156400040000  rdn:2.312321
T:70156400080000  rdn:2.140589 (ignored)
T:701564000100000 rdn:2.213509
                  rdn:1.915024 (others)
T:701564000160000 rdn:2.156720 (ignored)
```

Figure 6.17: 10% of data is collected from initial data flows.

(a)	(b)
T: 701557265200000	T: 701557098700000
T: 701557265220000	T: 701557098720000
T: 701557265240000	T: 701557098740000
T: 701557265260000	T: 701557098760000
T: 701557265280000	T: 701557098780000
T: 701557265300000	T: 701557098800000
T: 701557265320000	T: 701557098820000
T: 701557265340000	T: 701557098840000
T: 701557265360000	T: 701557098860000
T: 701557265380000	T: 701557098880000
T: 701557265400000	T: 701557098900000
T: 701557265420000	T: 701557098920000
T: 701557265440000	T: 701557098940000
T: 701557265460000	T: 701557098960000
T: 701557265480000	T: 701557098980000

Figure 6.18: The time patterns resulting from two types of monitoring

Two pieces of pseudo code written in accordance with the two methods of data collection are shown in the following:

(a)
```
if ( touch boundary ) {
  collect it;
  select next constraint if appropriate;
 }
```

(b)
```
collect it
```

The result comes largely from the method of maintaining a small-size default hierarchy. The filtering process is considerably shortened since most of the time only the top defaults are applied, and the attempt to alternate constraints is rarely tried more than twice. Though a small-size hierarchy of defaults does not affect the amount of data to be collected, it effectively controls the interference caused by excessive trials of alternatives. With these processes, there is little reason to doubt that the operating expense of running such an intelligent

monitor is close to the expense of running a software monitor designed with conventional technology.

3. Because the prototype carries out the knowledge refining process at a machine that is not part of the monitored system and is supported by a different file server, the considerable cost resulting from the refining does not bring significant interference. The only possible disturbance is that at a certain frequency or upon a signal received from the refining process when some changes have been made, the monitoring controller needs to retrieve updated criteria from buffers and to update the criterion tables. However, the implementation of a defining process in this prototype is only intended to show that such a refining process is capable of effectively changing the behavior of monitoring in terms of feedback from monitored data. Whether the performance of monitoring is improved through the refining process largely depends on the statistics package to be connected. One experiment of evolving defaults in the prototype is carried by selecting three heuristics—*most recently used, most anomalous,* and *oldest*—and by letting $a_1 = 0.34$, $a_2 = 0.33$, and $a_3 = 0.33$. After 250 data items are observed, the second coefficient set is adjusted to become $b_1 = 0.53$, $b_2 = 0.48$, and $b_3 = 0.49$. While three constraints are accepted for further filtering, the other four constraints are finally deleted, which is shown in Figure 3: The experiments demonstrate that the refining process is able to evolve defaults by following the specified methodology. Nevertheless, in recognition of the fact that, currently in this prototype, only limited knowledge of statistics is applied, such a evolution has no ground to promise that the refining process can lead to a better performance of monitoring; it merely shows that such possibility exists since the prototyped refining process illustrates a way of using an analytical package to update knowledge involved in monitoring.

4. The graphic display facilitated with multiple windows is also located remotely.

	constraint	since	0.3s	collected
case 1:	0.127692	467	103	21
case 2:	0.048350	249	22	5
case 3:	0.000000	9920	53	6
case 3:	0.113349	1338	11	5
case 3:	0.083254	797	11	3
case 3:	0.078951	104	31	1
case 3:	0.017043	928	11	4

Figure 6.19: The case evolvement in an experiment

This further reduces operating expenses. In addition, since only about 10% of observed data is to be sent, the cost of sending or buffering the collected data is largely reduced in comparison to a monitor with no selecting capability at the front stage.

Finally, while the prototype demonstrated the validity of the major components of the intelligent monitoring approach, further integration of these components for use in the real world still requires more work. Some improvements to be addressed in future research are discussed in Section 7.3. In spite of this, the detailed implementation of this prototype should only be a matter of time.

Chapter 7

Conclusions

The thesis proceeds to its conclusions. These conclusions consist of a brief summary, an emphasis on contributions, and the future work necessary for the completion of this research.

7.1 Summary

This dissertation has researched an intelligent approach to monitoring computer systems. In the researched monitoring model, a computer system or a computing activity is perceived at various levels of abstraction. These targets are further projected in terms of semantic concerns and are monitored with various resolutions. The resulting abstract representation is characterized by constraint objects, where primitive objects focus on the abstract behavior of individual computing units and virtual objects intend to reveal the coordination among those units. Such a representational model is dynamically defined and reshaped during monitoring.

Viewing a monitored system as a layered abstract model not only provides abstract views of system performance but also benefits the effectiveness of data collection. Within a reduced search space, default reasoning becomes a major controlling method

and brings advantages such as less computing time for reasoning while ensuring that only significant data is collected. With on-line controlling mechanisms, levels of detail, semantics, and resolution of monitoring can be controlled for effective performance of monitoring.

The integrated refining mechanisms provide the capability of improving the monitoring performance at run time. Defaults evolves during this process in order to compromise the unpredicted behaviors of monitored systems and enable the monitoring to operate in a more effective manner. Triggering methods and refining strategy are then adjusted to fit into the appropriate control over extended defaults.

The developed model for intelligent monitoring focuses on fundamental aspects of an intelligent system: knowledge representation, knowledge acquisition, and reasoning methods. Improvements in the first two aspects facilitate reasoning methods which can be carried out with less computation and better accuracy.

The resulting model has been prototyped, major mechanisms of this approach have been tested, and the anticipated properties of such a monitoring model have been found.

7.2 Contributions

The approach to intelligent monitoring developed in this thesis is significant and original. It addresses critical issues in monitoring modern computer systems primarily by enhancing the role of declarative knowledge in monitoring. As considered by Frederick Hayes-Roth *et al.* [38], monitoring systems should belong to one of six types of expert systems. However, as learned from the survey conducted in the preliminary study of this research, none of the existing monitoring projects have developed a concrete methodology for intelligent monitoring, even though many have been designed with this in mind. Dr. Snodgrass summarized his research [68] with

a strong indication about the necessity of intelligence in a monitoring system. He stated the following:

> An even more general ... future research is the extension of the knowledge base used by the monitor. ... Extensions to the knowledge base include representing causality in the monitor, and using the knowledge base to make inferences about future events.

The approach explored in this thesis advances monitoring technology and departs somewhat from currently existing approaches [79]. Consequent contributions of the researched model for intelligent monitoring can be abstracted into three aspects:

- The first distinct aspect is that the researched monitoring model can behave with remarkable flexibility in dealing with dynamic characteristics of modern computer systems. The representational model for a monitored system is reconfigurable and varies with abstraction and semantics; the behavior of monitoring is adaptable through updating constraints which stand for declarative knowledge; along with the monitoring, the underlying knowledge structure is extended and refined so that data collection is adjusted to achieve the desired effectiveness.

- The second notable achievement is that the model is capable of significantly reducing the amount of data collected while the comprehensibility and descriptiveness of monitoring are largely sustained. With abstract representations, the monitoring may focus on an abstract level. Only significant data is collected in terms of chosen constraints at a given level. By applying various constraints, the resolution of monitoring may be kept at an appropriate degree.

- The third expressive contribution is the point which shows that the resulting monitoring model can effectively facilitate performance analysis [66]. Only collecting meaningful information through the early-stage filtering process eases the generation of analytical data; this may avoid possible confusion resulting

from massive and noisy data. Next, being able to collect data at several levels simultaneously enables monitoring to provide analysts with a comprehensive view of a monitored system. The cluster method of recording data provides users with high-level information and may often make the later-stage simulation and modeling unnecessary. Furthermore, with the ability to autonomously adjust the details and scopes of monitoring in response to the occurrence of anomalies, the monitoring model may serve as a basis for diagnosis.

7.3 Future Work

The researched model has presented a fundamental framework for intelligent monitoring valid across machines. Much more research needs to be carried out before the model can be expected to produce intelligent monitors for practical use. This final section lists some future work for this research. Needed successive work which enlarges upon this research may be as follows.

Improving representational model A formal decomposition methodology, which addresses cases in which a monitored activity is neither sequentially nor vertically decomposable, is one of the extensions to the representational model. The situation in which two processes are mutually dependent—for example, calling each other—may preclude a strict hierarchy clearly derived from decomposition. A possible method is to maintain a more relaxed hierarchical structure. Such a hierarchy does not strictly depend on the logical relationship of activities. Rather, it may include a combination of physical, logical, and functional relations. This raises another issue of how to appropriately map between abstract layers which are situated in different relations.

Improving the maintenance of knowledge Declarative knowledge has played an important role in controlling monitoring. The knowledge base could grow immensely

so that it might require substantial computing time. Whether to leave this part of the management to be processed off-line should be considered. If it is needed, the next question involves which part should be retained for processing on-line and with what frequency the updating for consistency between two parts should be kept. Additionally, the capability for refining ought to be strengthened to ensure the evolvement of knowledge stability in a positive direction.

Extending the framework The research has initiated and substantiated an approach to intelligent monitoring. The resulting framework for intelligent monitoring is general and leaves a number of needed techniques undeveloped. They are, in fact, mandatory if such a model is to be applied in the real world. Among them are the extension of a sophisticated interface to users, a number of software probes for directly obtaining data from hardware probes, and communication media for coordinate monitoring.

In short, the basis for an approach to intelligent monitoring has been established. Continued research in this area should greatly alleviate the burdensome challenges that lie ahead in monitoring modern computer systems.

Appendix A

Conventional Approaches to Monitoring

Hardware Monitoring Approach

Traditionally, monitoring and measurement have been done with fundamental techniques in hardware engineering [17, 1]. A hardware monitor is a device that is not part of a monitored system. For a typical hardware monitor, probing into the inner workings of a machine to accumulate data should make it possible to set key system parameters to optimal values. One of the main advantages of hardware monitoring is that such a device can be designed to have minimal or no effect on a host system. This method fits well into some aspects of real-time monitoring. Currently, commercial hardware monitors are widely available for measuring systems' performance and tuning primarily large multiprogramming installations. However, hardware monitoring does not solve monitoring problems regarding users' concerns. The reasons for stating this idea are several.

First, hardware monitors generally provide only limited, low-level information about activities of a host system. Those monitors have already reached the frontiers

of measurability in modern computer systems: simple observation of system buses, or probes connected to the processor and memory ports and I/O channels; nevertheless, they are restricted to a few fixed observation points and provide a low-level interface as well as low-level data.

Secondly, these monitors often use sophisticated features of the hardware to get valuable information. The use of these monitors is generally restricted to experts. Their installation requires much expertise and a thorough understanding of a system.

Third, since valuable information is obtained by relying on a fixed address, hardware monitors can not handle either the dynamic creation and the deletion of monitoring processes, or the observation of the migration of program parts in memory and the use of memory management units.

Software Monitoring Approach

Recently, the increased complexity of present computer systems necessitates monitoring tools for software development [18]. Software monitors can present information in an application-oriented manner. These monitors are usually contained within the measured system, sharing the same execution environment, thus producing some degree of interference in both the timing and space of the monitored program. The major deficiencies with the software monitoring approach are overhead, inaccuracy and performance degradation, and change of system behavior. However, many software monitoring facilities are still playing a major role in monitoring [2]. The reasons are these: 1) it is easy to adapt to different monitoring granularities; 2) it allows users to interactively evaluate the performance history of system activities; 3) it usually can be made portable across different types of machines.

Hybrid Methodology of Monitoring

The main advantage of adopting a hybrid model is being capable of overcoming the deficiencies if only the hardware or software approach is used. Monitoring facilities that are completely implemented in hardware are nonperturbing and can obtain architectural performance views, but they are electrically complex and costly and do not have a "view" of the software structures that they are measuring. To take advantage of this, a careful choice of the support from hardware may greatly reduce monitoring perturbation but at a modest cost [53, 34]. The monitoring software aspect can be responsible for filtering incoming information, collecting high-level information, and converting data flows for display and further analysis. Due to these advantages, the hybrid approach has been drawing more interest.

Appendix B

Understanding of Object Orientation

Frame–The Precedence of Object Orientation

The technology of object orientation is derived from the concept of *frame* that was initially introduced in 1975 by Marvin Minsky [56]. In his definition, a *frame* is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is concerned with how to use the frame. Some is about what one can expect to happen next. Other kinds deal with solutions if these expectations are not confirmed. Collections of related frames are linked together into *frame-systems.* The effects of important actions are mirrored by *transformations* between the frames of a system. These are used to make certain kinds of calculations economical, to represent changes of emphasis and attention, and to account for the effectiveness of imagery.

The evolution of frame representation was shaped by the goal of creating data structures that are, in a useful manner, "the same as" the world they represent.

This is the property of *homomorphism*: the representation is a "direct" image of the world it is intended to describe; it allows programmers to think about the formal representations in the knowledge base in the same terms as they think about objects in the domain.

Object Orientation

The deficiency of frame structure is its passiveness, that is, itself cannot be not a computational object. The basis of object-oriented model is the ability to define computational objects with arbitrary complex internal structures, which may be thought of as a single entity [49]. They are active in the sense that the methods are bound to the object itself, rather than existing as separate procedures for the manipulation of a data structure. Outlined in [12, 75], the main object-oriented characteristics are:

- Encapsulation (data abstraction): An object consists of some variables and the allowable operations (known as methods) on them. The value of these variables cannot be changed directly. They can only be changed by sending a message to the appropriate method in the object. Furthermore, objects with the same methods and definition of variables are grouped into one object type (in some object systems, the term "class" is used instead of "type").

- Independence: Objects have control over their own state (i.e., value of the variables) and existence (i.e., continual existence even if its creator dies).

- Message-passing paradigm: Objects cooperate (or interact with each other) by passing messages. If there is no more than one object executing at any time, then the message passing is similar to a procedure call in traditional programming languages.

- Inheritance: A technique that allows new classes to be built on top of the older, less specialized classes (instead of building them from scratch). A new class is

created out of old ones by specifying how the new one differs from the old. This is the same concept as the specialization.

- Homogeneity: In a pure object-oriented language, everything is an object. For example, the number "3" is an object, a message is an object, etc.

- Concurrence: Every object can be an active entity, since each object has its own methods, variables, communicating medium, etc. In addition, each can have a certain long lift time. These features make an object functionally independent.

The properties of object orientation support a powerful and natural way to organize large and complex software implementations and are equally applicable in the design of those systems integrated with artificial intelligence.

Finally, numerous advantages can be achieved by means of object-oriented approaches to building an expert system. It appears evident that the best place to introduce the use of the object-objected approach is at a level where the inference engine and knowledge representation schemes are implemented.

Appendix C

Simulating Targets

Targets are provided through a simulation generator. The generator is situated at an abstract mathematical model and simulates such systems whose behaviors exhibit typical characteristics of modern computer systems. To mimic actual computing activities, a simulating model can be as complicated as a designer wants. The strategy adopted here is to generate typical data which simulates such complicated activities that they are likely decomposed and understood at different levels.

Generated data is organized in groups to represent necessary types, levels, semantics, and patterns of data which are expected to be confronted in monitoring a real system. The rest of this appendix briefly examines how such data is simulated.

Primitive Random Data The origination of different simulated data is initially developed through functions generating random numbers. The following is the code for this function. The random number function consists of two routines, The first one generates a short sequence of number by a linear generating function. Then, to improve the random degree, uses additive congruent method to extend the initial sequence. The improvement of randomness counts on the length of a base sequence used in the second routine.

```
unsigned long A, C, Z, seed, base[INT_LENGTH];
unsigned long aij, M;

unsigned long linear()
{
    Z = (A * Z + C) % M;
    return((unsigned long) Z );
}


double addrdn()
{
    unsigned long temp;
    double temp;

    temp = (aij*(base[0]+base[1]+base[2]))%M;
    base[ptr] = temp;
    ptr = (ptr+1)%K;
    return (((double)temp)  ((double)M));
}
```

Figure C.1: The random number generator based on an additive congruent method.

```
long *types_data();
{
  data = addrdn();
  switch ( mapping_to_data(data) ) {
    case STRING:
        return get_string(data);
    case CHARACTER:
        return get_char(data);
    case DIGITS:
        return get_digits(data);
    case INTEGER:
        return get_integer(data);
    case BOOLEAN:
        return get_boolean(data);
    default:
        return UNKNOWN;
  }
}
```

Figure C.2: The function that generates various types of data.

Types of Data With the basis of the above random data, one type of generation is to simulate data in regard to different types. The simulated types include integer, boolean, digital number, and characters. Furthermore, the integer type of data is again divided into several groups representing diverse semantic meaning. The code in Figure C.2 accomplishes the generation of various types of data.

Levels of Data Another concern of generation is to produce data that may be considered to occur at different levels. Types of data are distinguished through patterns, frequencies, and ranges. In relation to other groups of data that are purposefully produced at a low rate, some groups of data are designed to be produced proportionally more so that the data may be thought of as occurring at lower levels. About three to four levels of data are simulated by the routine shown in Figure C.3.

```
double level_data( );
{
  double data;

  data = addrdn();
  switch ( mapping_to_level(data) ) {
    case 1:
        return mapping_to_1(data);
    case 2:
        return mapping_to_2(data);
    case 3:
        return mapping_to_3(data);
    case 4:
        return mapping_to_4(data);
    default:
        return UNKNOWN;
  }
}
```

Figure C.3: Mapping data into groups and levels.

Semantics of Data Data is also distinguished in light of semantics. The data for simulating events and data flows are differently marked. A data record consisting of a head and a body is used to simulate an event. The head part is used to decide which type of event, while the body part presents further details of this event. The data for simulating data flows consists of two parts that are recognized as a pointer and as the data body itself. Figure C.4 displays the code carrying out this function.

Patterns of Data The first pattern of sequences of generated numbers is a uniform distribution. This pattern in fact is often seen in practise and can be generated through a random generator and some routines which convert data to different meanings. Another large category of sequences is that of nonuniform distribution. Two main nonuniform patterns are used in the implementation. Where one is the exponentially distributed numbers, another is geometrically distributed numbers. The basic code invoked in generating these two patterns is listed in Figure C.5. The

```
double semantic_data( ptrSD )
int *ptrSD;
{
  double data;
  int semanticData;

  data = addrdn();
  switch ( take_head(data) ) {
    case EVENT:
        semanticData = EVENT;
        return pick_tail(data);
    case DATA_FLOW:
        semanticData = DATA_FLOW;
        return pick_tail(data);
    default:
        return UNKNOWN;
  }
}
```

Figure C.4: Generating semantics-related data.

above patterns are further customized to serve as describing computing activities. Figure C.6 lists the code that simulates a pattern depicting a customer queue. This pattern results from a queuing policy: a job having a higher priority is served first; it is assumed that no breaking in is allowed; that is, once a job gets service it will not be interrupted until finished or exited due to fatal errors. A parameter passed into the routine at the time it is called is a uniformly random number. This indicates the arrival pattern of jobs with a priority that is assigned also based on a series of random numbers.

Independence of Simulated Data Part of the effort contributed to the simulation is to check the independence of these random numbers. By using the additive congruent method, the generator avoids inadequate long cycles such as 2^w, where 2^w is a modulo if a linear congruent method is used. Instead, the period may reach as long as m^k - 1, where m is a modulo used in generation and k is the length of a

```
double exprdn()
{
    double temp;

    temp = addrdn();
    if (temp > P) {
      temp = (temp-P)/(1-P);
      return (-log(temp)/u2);
    }
    else {
      temp = temp/P;
      return (-log(temp)/u1);
    }
}

 double geom( possib )
 int possib;
 {
     int n = 1;

     Z = addrdn();
     while ( Z > possib ) {
         n++;
         Z = addrdn();
     }
 }
```

Figure C.5: Simulating two nonuniform patterns.

```
int serving_with_priority(float bound)
    /* bound indicates a priority.  */
{
    int n = 1;

    z := addrdn();
    while ( z < bound ) {
    * if true, then means no jobs come yet *
        n++;
        z = addrdn();
    }     return( n );
}
```

Figure C.6: The serving function that follows a priority principle.

sequence of numbers. Choosing a sequence of 10 numbers and a modulo as 2^8, the resulting generation of random numbers achieves a degree of freedom sufficiently high so that a confidence of more than 99% is reached.

Bibliography

[1] B. Ableidinger, N. Agarwal, and C. Nobles. Real-time analyzer furnishes high-level look at software operation. *Electronic Design*, pages 117–131, September 1985.

[2] ACM. *Proc. of the ACM Symposium on High Level Debugging.* SIGPLAN, 1983.

[3] B. Bates and J. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Systems and Software*, pages 255–264, March 1983.

[4] P. Bates and J. Wileden. An approach to high-level debugging of distributed systems. *Softw. Eng.*, 8(4):107, August 1983.

[5] Philippe Besnard. *An Introduction to Default Logic.* Springer-Verlag, pages 37-74, 1989.

[6] L. Bic. *The Logical Design of Operating Systems.* Prentice Hall, 1986.

[7] A. Borning. Defining constraints graphically. Univ. of Washington Comp. Scien. Dept. Technical Report No. 85-09-05, Seattle, Wash, 1985.

[8] D. Buck and D. Hrynyk. Software architecture for a computer network monitoring system. In *Performance of Computer Installations*, 1978.

[9] J. Carbonell and Y. Gil. *Learning by Experimentation: The Operator Refinement Method.* Morgan Kaufmam, 1990.

[10] K. Chandy and L. Lamport. Distributed sanpshort: Determining global states of distributed systems. *ACM Trans. Computer Systems*, 3(1):63–75, 1985.

[11] W. Clancey. The advantages of abstract control knowledge in expert system design. *AAAI*, 3, 1983.

[12] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press Computing Series, 1990.

[13] Digital Equipment Corporation. *VAX Software Handbook.* Maynard, Mass., 1982.

[14] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 4:347–410, 1984.

[15] J. Diederich and J. Milton. Objects, messages, and rules in database design. In *Object-Oriented Concepts, Databases, and Applications*, pages 177–197, 1989.

[16] B. Duval and Y. Kodratoff. Automated deduction in an uncertain and inconsistent data basis. In *ECAI, Brighton*, pages 101–108, 1986.

[17] A. Mink *et al.* Hardware-assisted multiprocessor performance measurements. In *Int'l Symp. Performance: Performance 87*, pages 151–168, 1987.

[18] A. Mink *et al.* Multiprocessor performance-measurement instrumentation. *IEEE Computer*, pages 69–73, September 1990.

[19] B. Miller *et al.* A distributed programs monitor for berkeley unix. *Softw. Practice and Experience*, 16(2):183–200, February 1986.

[20] H. Garcia-Molina *et al.* Debugging a distributed computing system. *IEEE Trans. Softw. Eng.*, 10(2):210, March 1984.

[21] H. Tokuda *et al.* A real-time monitor for a distributed real-time operating system. *IEEE Trans. on SE*, pages 68–77, January 1989.

[22] J. Joyce *et al.* Monitoring distributed systems. *ACM Trans. on Comp. Syst.*, 5(2):121–150, May 1987.

[23] J. Joyce *et al.* Monitoring distributed systems. *ACM Trans. on Comp. Syst.*, 5(2):122, May 1987.

[24] M. Brodie *et al.* Future artificial intelligence requirements for intelligent database systems. In *Proc. of the 2nd Int. Conf. on Expert Database Systems*, pages 45–62, 1988.

[25] M. McKusick *et al.* A fast file system for unix. In *Unix System Manager's Manual*, page 14, 1986.

[26] M. Morgenstern *et al.* Constraint-based systems: knowledge about data. In *Proc. of the 2nd Int. Conf. on Expert Database Systems*, pages 23–44, 1988.

[27] U. Dayal *et al.* Simplifying complex objects: The probe approach to modeling and querying. In *Readings in Object-Oriented Database Systems*, pages 390–399, 1990.

[28] D. Ferrari, C. Serazzi, and A. Zeigner. *Measurement and Tuning of Computing Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[29] C. Froidevaux. Taxonomic default theory, brighton. In *ECAI*, pages 123–129, 1986.

[30] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, pages 152-155, 1990.

[31] J. Giarratano and G. Riley. *Expert Systems, Principles and Programming.* PWS-KENT Publishing Company, Boston, 1989.

[32] J. Gosling. 1983. CMU Technical Report CS-83-132 (Ph.D. thesis), Carnegie-Mellon University, Pittsburg, Algebraic Constraints.

[33] D. Haban and D. Wybranietz. Hardware supported monitoring in distributed computer systems. SFB124 Report, No. 23/86, University of Kaiserslautern, 1986.

[34] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Trans. on Softw. Eng.*, pages 198–200, February 1990.

[35] S. Hanson and D. Burr. Knowledge representation in connectionist networks. Bellcore Technical Report, 1987.

[36] F. Harmelen. A classification of meta-level architectures. In *Logic-Based Knowledge Representation*, pages 13–35, 1989.

[37] P. Hayes. The logic of frames. In *Frame Conceptions and Text Understanding*, pages 46–61, 1979.

[38] F. Hayes-Roth. *Building Expert Systems.* Addison-Wesley, pages 13-16, 1983.

[39] J. Hendler and C. Lewis. Designing interfaces for expert systems. In *Expert Systems: The User Interface*, pages 1–15, 1988.

[40] J. Joyce and B. Unger. Graphical monitoring of distributed systems. In *SCS conference on AI, Graphics and Simulation, San Diego, Calif.*, pages 85–92, 1985.

[41] T. Kerola and H. Schwetman. Monit: A performance monitoring tools for parallel and pseudo-parallel programs. *ACM Machinery*, pages 163–173, May 1987.

[42] D. Klahr, P. Langley, and R. Neches. *Production System Models of Learning and Development.* MIT Press, 1987.

[43] J. Laird, P. Rosenbloom, and A. Newell. Towards chunking as a general learning mechanism. Technical Report cmu-cs-85-110, Dept. of Computer Science, CMU, 1985.

[44] B. Lazzerini and L. Lopriore. Abstraction mechanisms for event control in program debugging. *IEEE Transaction on SE*, 15(7):890–901, July 1989.

[45] W. Leler. Constraint languages for computer aided design. *SIGDA Newsletter*, 15(2):11–15, June 1985.

[46] H. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2), 1984.

[47] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *the first ACM conference on object-oriented programming systems, languages and applications, SIGPLAN Notices 21-9*, pages 214–223, 1986.

[48] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in clu. In *Readings in Object-Oriented Database Systems*, pages 47–58, 1990.

[49] G. Luger and W. Stubblefield. *Artificial Intelligence and the Design of Expert Systems.* Library of Cong. Cataloging-in-Publishing Data, 1989.

[50] W. Lukaszewicz. Nonmonotonic logic for default theories. In *ECAI*, pages 305–314, 1984.

[51] W. Lukaszewicz. Two results on default logic. In *IJCAI, Los Angeles*, pages 459–461, 1985.

[52] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[53] S. Melvin and Y. Patt. Monitoring and performance measuring distributed systems during operation. *ACM Trans. Computer Systems*, March 1988.

[54] R. Michalski. Knowledge acquisition through conceptual clustering: A theoretical framework and an algorithm for partitioning data into conjunctive concepts. *IJ of Policy Analysis and Information Systems*, 4:219–244, 1980.

[55] B. Miller and C. Yang. Ipc: An interactive and automatic performance measurement tool for parallel and distributed systems. In *Proc. 7th Int. Conf. on Distributed Computing Systems*, pages 482–489, 1987.

[56] M. Minsky. A framework for representing knowledge. *The Psychology of Computer Vision, P.H. Winston. Ed. McGraw-Hill, New York*, 1975.

[57] M. Model. Monitoring system behavior in a complex computational environment. Ph.D. Dissertation, Stanford University, Standford, California, 1978.

[58] N. Nounou and Y. Yemini. Development tools for communication protocols. Res. Report, CUCS-160-85, Dept. of Computer Science, Columbia University, 1985.

[59] B. Plattner and J. Nievergelt. Monitoring program execution: A survey. *IEEE Computer*, pages 80–81, November 1981.

[60] X. Qian. Distributed design of integrity constraints. In *Proc. of the 2nd Int. Conf. on Expert Database Systems*, pages 205–226, 1988.

[61] R. Reiter. On reasoning by default, urbana, ill. In *Theoretical Issues in Natural Language Processing*, pages 210–218, 1978.

[62] E. Rich. Default reasoning as likelihood reasoning. *Int. Jour. Computers and Mathematics, Special Issues On Computational Linguistics*, 9(1):1–13, 1983.

[63] D. Russinoff. Proteus: A frame-based nonmonotomic inference system. In *Object-Oriented Concepts, Databases, and Applications*, pages 128–132, 1989.

[64] M. Ryszard and K. Yves. *Research in Machine Learning.* Morgan Kaufmam, 1990.

[65] M. Sachais. *VSAM Tuning and Advanced Topics.* Van Nostrand Reinhold, New York, 1989.

[66] S. Shen, R. Mukkamala, and M. Xu. A monitoring model as the basis for performance analysis. In *Proc. of the ISMM International Conf. on Computer Applications in Design, Simulation, and Analysis*, pages 223–226, 1992.

[67] Y. Shoham. Temporal logics in ai: Semantical and ontological considerations. *Artificial Intelligence*, pages 89–104, 1987.

[68] R. Snodgrass. Monitoring distributed systems: A relational approach. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1982.

[69] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Trans. on Comp. Syst.*, 6(2):157–196, May 1988.

[70] L. Stein. Delegation is inheritance. In *Proc. of 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 138–146, 1987.

[71] L. Stein, H. Lieberman, and D. Ungar. A shared view of sharing the treaty of orlando. In *Object-Oriented Concepts, Databases, and Applications*, pages 31–48, 1989.

[72] D. Stemple, T. Sheard, and R. Bunker. Abstract data types in databases: Specification, manipulation and access. In *Readings in Object-Oriented Database Systems*, pages 345–351, 1990.

[73] M. Stonebraker and M. Hearst. Future trends in expert database systems. In *Proc. of the 2nd Int. Conf. on Expert Database Systems*, pages 3–20, 1988.

[74] L. Svobodova. Performance monitoring in computer systems: A structure approach. *Operating Systems Review*, 15(30):39–42, July 1981.

[75] E. Tello. *Object-Oriented Programming for Artificial Intelligence: A Guide to Tools and System Design.* Addison-Wesley Publishing Company, Inc., 1990.

[76] D. Touretzky. Implicit ordering of defaults in inheritance systems. In *AAAI, Austin*, pages 322–325, 1984.

[77] T. Winograd. Frame representation and the declarative/procedual controversy. In *Representation and Understanding: Studies in Cognitive Science*, pages 185–210, 1975.

[78] C. Wu. A hierarchical knowledge based system for airplane recognition. Ph.D. Dissertation, Dept. Elec. Eng. Syst., Univ. Southern California, 1987.

[79] M. Xu, S. Shen, and R. Mukkamala. An approach to intelligent monitoring. In *Proc. of the IEEE/ACM Int. Conf. on Developing and Managing Expert System Programs*, pages 178–186, 1991.

[80] J. Zhu and D. Maier. Abstract objects in an object-oriented data model. In *Proc. of the 2nd Int. Conf. on Expert Database Systems*, pages 73–106, 1988.

AUTOBIOGRAPHICAL STATEMENT

Myron Z. Xu received his BS in computer application from Shanghai Industry University in 1982 and his MS in computer science from Brigham Young University in 1989. During his fourteen year career in computer science, he has received numerous appointments as a teaching assistant, research assistant, software engineer, and full-time/part-time instructor.

Through his active participation in a number of research projects, he has published several research papers in collaboration with colleagues. Among them are "A Monitoring Model as the Basis for Performance Analysis," (Proceedings of the International Society for Mini and Microcomputers International Conference in Design, Simulation, and Analysis, 1992); "An Approach to Intelligent Monitoring," (Proceedings of the IEEE/ACM International Conference on Developing and Managing Expert System Programs, 1991); "Intelligent Monitoring: An Effective Way to Facilitate Performance Analysis," (Proceedings of the Conference on Modeling and Simulation, Vol. 22, No. 2, 1991); and "A Flexible Hypermedia Tool for Information Management," (Proceedings of the Symposium of AI Applications for Military Logistics, American Defense Preparedness Association, 1991).

He has a lifetime membership in the Golden-Key National Honor Society and is also a member of the Association for Computing Machinery and the SIGMETRICS (Measurement & Evaluation).